An implementation of ROBDDs for Isabelle/HOL

Julius Michaelis and Maximilian Haslbeck and Peter Lammich and Lars Hupel

March 17, 2025

Abstract

We present a verified and executable implementation of ROBDDs in Isabelle/HOL. Our implementation relates pointer-based computation in the Heap monad to operations on an abstract definition of boolean functions. Internally, we implemented the if-then-else combinator in a recursive fashion, following the Shannon decomposition of the argument functions. The implementation mixes and adapts known techniques and is built with efficiency in mind.

Contents

1 Preface

This work is not the first to deal with BDDs in Isabelle/HOL. Ortner and Schirmer have formalized BDDs in [4] and proved the correctness of an algorithm that transforms arbitrary BDDs to ROBDDs. However, their specification does not provide efficiently executable algorithms on BDDs. Giorgino and Strecker have presented efficiently executable algorithms for ROBDDs [2] by reducing their arguments to manipulating edges of graphs. However, they have, to the best of our knowledge, not made their theory files available. Thus, no library for efficient computation on (RO)BDDs in Isabelle/HOL existed. Our work is a response to that situation.

The theoretic background of the implementation is mostly based on [1].

2 Boolean functions

theory *Bool-Func*
imports *Main*
begin

The end result of our implementation is verified against these functions:

type-synonym *'a boolfunc* = (*'a* ⇒ *bool*) ⇒ *bool*

if-then-else on boolean functions.

definition *bf-ite i t e* ≡ (λ*l*. *if i l then t l else e l*)

if-then-else is interesting because we can, together with constant true and false, represent all binary boolean functions using maximally two applications of it.

abbreviation *bf-True* ≡ (λ*l*. *True*)
abbreviation *bf-False* ≡ (λ*l*. *False*)

A quick demonstration:

definition *bf-and a b* ≡ *bf-ite a b bf-False*
lemma (*bf-and a b*) *as* ⟷ *a as* ∧ *b as* **unfolding** *bf-and-def bf-ite-def* **by** *meson*

definition *bf-not b* ≡ *bf-ite b bf-False bf-True*
lemma *bf-not-alt*: *bf-not a as* ⟷ ¬*a as* **unfolding** *bf-not-def bf-ite-def* **by** *meson*

For convenience, we want a few functions more:

definition *bf-or a b* ≡ *bf-ite a bf-True b*
definition *bf-lit v* ≡ (λ*l*. *l v*)
definition *bf-if v t e* ≡ *bf-ite* (*bf-lit v*) *t e*

lemma *bf-if-alt*: *bf-if v t e* = (*λl. if l v then t l else e l*) **unfolding** *bf-if-def bf-ite-def bf-lit-def* **..**
definition *bf-nand a b* = *bf-not* (*bf-and a b*)
definition *bf-nor a b* = *bf-not* (*bf-or a b*)
definition *bf-biimp a b* = (*bf-ite a b* (*bf-not b*))
lemma *bf-biimp-alt*: *bf-biimp a b* = (*λl. a l* ⟷ *b l*) **unfolding** *bf-biimp-def bf-not-def bf-ite-def* **by**(*simp add*: *fun-eq-iff*)
definition *bf-xor a b* = *bf-not* (*bf-biimp a b*)
lemma *bf-xor-alt*: *bf-xor a b* = (*bf-ite a* (*bf-not b*) *b*)
unfolding *bf-xor-def bf-biimp-def bf-not-def*
unfolding *bf-ite-def*
by *simp*

All of these are implemented and had their implementation verified.

definition *bf-imp a b* = *bf-ite a b bf-True*
lemma *bf-imp-alt*: *bf-imp a b* = *bf-or* (*bf-not a*) *b* **unfolding** *bf-or-def bf-not-def bf-imp-def* **unfolding** *bf-ite-def* **unfolding** *fun-eq-iff* **by** *simp*

lemma [*dest*!,*elim*!]: *bf-False* = *bf-True* ⟹ *False bf-True* = *bf-False* ⟹ *False* **unfolding** *fun-eq-iff* **by** *simp-all*

lemmas [*simp*] = *bf-and-def bf-or-def bf-nand-def bf-biimp-def bf-xor-alt bf-nor-def bf-not-def*

2.1 Shannon decomposition

A restriction of a boolean function on a variable is creating the boolean function that evaluates as if that variable was set to a fixed value:

definition *bf-restrict* (*i*::′*a*) (*val*::*bool*) (*f*::′*a boolfunc*) ≡ (*λv. f* (*v*(*i*:=*val*)))

Restrictions are useful, because they remove variables from the set of significant variables:

definition *bf-vars bf* = {*v*. ∃ *as*. *bf-restrict v True bf as* ≠ *bf-restrict v False bf as*}
lemma *var* ∉ *bf-vars* (*bf-restrict var val ex*)
unfolding *bf-vars-def bf-restrict-def* **by**(*simp*)

We can decompose calculating if-then-else into computing if-then-else of two triples of functions with one variable restricted to true / false. Given that the functions have finite arity, we can use this to construct a recursive definition.

lemma *brace90shannon*: *bf-ite F G H ass* =
bf-ite (*λl. l i*)
(*bf-ite* (*bf-restrict i True F*) (*bf-restrict i True G*) (*bf-restrict i True H*))
(*bf-ite* (*bf-restrict i False F*) (*bf-restrict i False G*) (*bf-restrict i False H*))
ass
unfolding *bf-ite-def bf-restrict-def* **by** (*auto simp add*: *fun-upd-idem*)

end

3 Binary Decision Trees

theory *BDT*
imports *Bool-Func*
begin

We first define all operations and properties on binary decision trees. This has the advantage that we can use a simple, structurally defined type and the disadvantage that we cannot represent sharing.

datatype *'a ifex* = *Trueif* | *Falseif* | *IF 'a 'a ifex 'a ifex*

The type is the same as in Boolean Expression Checkers by Nipkow [3]. Internally, Boolean Expression Checkers transforms the boolean expressions to reduced BDTs of this type. Tests like being tautology testing are then trivial.

fun *val-ifex* :: *'a ifex* ⇒ (*'a* ⇒ *bool*) ⇒ *bool* **where**
  val-ifex Trueif s = *True* |
  val-ifex Falseif s = *False* |
  val-ifex (*IF n t1 t2*) *s* = (*if s n then val-ifex t1 s else val-ifex t2 s*)

fun *ifex-vars* :: (*'a* :: *linorder*) *ifex* ⇒ *'a list* **where**
  ifex-vars (*IF v t e*) = *v* # *ifex-vars t* @ *ifex-vars e* |
  ifex-vars Trueif = [] |
  ifex-vars Falseif = []

abbreviation *ifex-var-set a* ≡ *set* (*ifex-vars a*)

fun *ifex-ordered* :: (*'a*::*linorder*) *ifex* ⇒ *bool* **where**
  ifex-ordered (*IF v t e*) = ((∀ *tv* ∈ (*ifex-var-set t* ∪ *ifex-var-set e*). *v* < *tv*)
    ∧ *ifex-ordered t* ∧ *ifex-ordered e*) |
  ifex-ordered Trueif = *True* |
  ifex-ordered Falseif = *True*

fun *ifex-minimal* :: (*'a*::*linorder*) *ifex* ⇒ *bool* **where**
  ifex-minimal (*IF v t e*) ⟷ *t* ≠ *e* ∧ *ifex-minimal t* ∧ *ifex-minimal e* |
  ifex-minimal Trueif = *True* |
  ifex-minimal Falseif = *True*

abbreviation *ro-ifex* **where** *ro-ifex t* ≡ *ifex-ordered t* ∧ *ifex-minimal t*

definition *bf-ifex-rel* **where**
  bf-ifex-rel = {(*a*,*b*). (∀ *ass*. *a ass* ⟷ *val-ifex b ass*) ∧ *ro-ifex b*}

lemma *ifex-var-noinfluence*: *x* ∉ *ifex-var-set b* ⟹ *val-ifex b* (*ass*(*x*:=*val*)) = *val-ifex b ass*
  by (*induction b*, *auto*)

lemma *roifex-var-not-in-subtree*:
  assumes *ro-ifex b* **and** *b* = *IF v t e*

shows *v* ∉ *ifex-var-set t* **and** *v* ∉ *ifex-var-set e*
using *assms* **by** (*induction*, *auto*)

lemma *roifex-set-var-subtree*:
assumes *ro-ifex b* **and** *b* = *IF v t e*
shows *val-ifex b* (*ass*(*v*:=*True*)) = *val-ifex t ass*
val-ifex b (*ass*(*v*:=*False*)) = *val-ifex e ass*
using *assms* **by** (*auto intro*!: *ifex-var-noinfluence dest*: *roifex-var-not-in-subtree*)

lemma *roifex-Trueif-unique*: *ro-ifex b* ⟹ ∀ *ass*. *val-ifex b ass* ⟹ *b* = *Trueif*
proof(*induction b*)
case (*IF v b1 b2*) **with** *roifex-set-var-subtree*[*OF* ‹*ro-ifex* (*IF v b1 b2*)›] **show** *?case* **by** *force*
qed(*auto*)

lemma *roifex-Falseif-unique*: *ro-ifex b* ⟹ ∀ *ass*. ¬ *val-ifex b ass* ⟹ *b* = *Falseif*
proof(*induction b*)
case (*IF v b1 b2*) **with** *roifex-set-var-subtree*[*OF* ‹*ro-ifex* (*IF v b1 b2*)›, *of v b1 b2*] **show** *?case*
by *fastforce*
qed(*auto*)

lemma (*f*, *b*) ∈ *bf-ifex-rel* ⟹ *b* = *Trueif* ⟷ *f* = (λ-. *True*)
unfolding *bf-ifex-rel-def* **using** *roifex-Trueif-unique* **by** *auto*

lemma (*f*, *b*) ∈ *bf-ifex-rel* ⟹ *b* = *Falseif* ⟷ *f* = (λ-. *False*)
unfolding *bf-ifex-rel-def* **using** *roifex-Falseif-unique* **by** *auto*

lemma *ifex-ordered-not-part*: *ifex-ordered b* ⟹ *b* = *IF v b1 b2* ⟹ *w* < *v* ⟹ *w* ∉ *ifex-var-set b*
using *less-asym* **by** *fastforce*

lemma *ro-ifex-unique*: *ro-ifex x* ⟹ *ro-ifex y* ⟹ (⋀*ass*. *val-ifex x ass* = *val-ifex y ass*) ⟹ *x* = *y*
proof(*induction x arbitrary*: *y*)
case (*IF xv xb1 xb2*) **note** *IFind* = *IF*
from ‹*ro-ifex* (*IF xv xb1 xb2*)› ‹*ro-ifex y*› ‹⋀*ass*. *val-ifex* (*IF xv xb1 xb2*) *ass* = *val-ifex y ass*›
show *?case*
proof(*induction y*)
case (*IF yv yb1 yb2*)
obtain *x* **where** *x-def*: *x* = *IF xv xb1 xb2* **by** *simp*
obtain *y'* **where** *y'-def*: *y'* = *IF yv yb1 yb2* **by** *simp*
from *y'-def x-def IFind IF* **have** *0*: *ro-ifex xb1 ro-ifex xb2 ro-ifex yb1*
ro-ifex yb2 ro-ifex x ro-ifex y' **by** *auto*
from *IF IFind x-def y'-def* **have** *1*: ⋀*ass*. *val-ifex x ass* = *val-ifex y' ass* **by** *simp*
show *?case*
proof(*cases xv* = *yv*)

case *True*
have *xb1* = *yb1*
by (*auto intro*: *IFind simp add*: *0 1 True roifex-set-var-subtree*[*OF* - *y'-def*]
roifex-set-var-subtree[*OF* - *x-def*, *symmetric*])
moreover have *xb2* = *yb2*
by (*auto intro*: *IFind simp add*: *0 1 True roifex-set-var-subtree*[*OF* - *y'-def*]
roifex-set-var-subtree[*OF* - *x-def*, *symmetric*])
ultimately show *?thesis* **using** *True* **by** *simp*
next
case *False* **note** *uneq* = *False* **show** *?thesis*
proof(*cases xv* < *yv*)
case *True*
from *ifex-ordered-not-part*[*OF* - *y'-def True*] *ifex-var-noinfluence*[*of xv y'* - *True*]
0(*6*) *roifex-set-var-subtree*(*1*)[*OF 0*(*5*) *x-def*] *1*
have *5*: ⋀*ass*. *val-ifex xb1 ass* = *val-ifex x ass* **by** *blast*
from *0*(*5*) *ifex-var-noinfluence*[*of xv xb1* - *False*]
ifex-var-noinfluence[*of xv xb2* - *False*]
x-def
have ⋀*ass*. *val-ifex xb1* (*ass*(*xv* := *False*)) = *val-ifex xb1 ass*
⋀*ass*. *val-ifex xb2* (*ass*(*xv* := *False*)) = *val-ifex xb2 ass* **by** *auto*
from *5 this roifex-set-var-subtree*(*2*)[*OF 0*(*5*) *x-def*]
have ⋀*ass*. *val-ifex xb1 ass* = *val-ifex xb2 ass* **by** *presburger*
from *IFind*(*1*)[*OF 0*(*1*) *0*(*2*)] *this IFind*(*3*) **have** *False* **by** *auto*
from *this* **show** *?thesis* **..**
next
case *False*
from *this uneq* **have** *6*: *yv* < *xv* **by** *auto*
from *ifex-ordered-not-part*[*OF* - *x-def this*]
ifex-var-noinfluence[*of yv x*] *0*(*5*)
have ⋀*ass val*. *val-ifex x* (*ass*(*yv* := *val*)) = *val-ifex x ass*
⋀*ass val*. *val-ifex x* (*ass*(*yv* := *val*)) = *val-ifex x ass* **by** *auto*
from *this roifex-set-var-subtree*[*OF 0*(*5*) *x-def*]
have ⋀*ass val*. *val-ifex x* (*ass*(*xv* := *True*, *yv* := *val*)) = *val-ifex xb1 ass*
⋀*ass val*. *val-ifex x* (*ass*(*xv* := *False*, *yv* := *val*)) = *val-ifex xb2 ass* **by** *blast*+
from *ifex-ordered-not-part*[*OF* - *x-def 6*] *0*(*5*) *ifex-var-noinfluence*[*of yv x*] *1*
roifex-set-var-subtree[*OF 0*(*6*) *y'-def*]
have ⋀*ass val*. *val-ifex x ass* = *val-ifex yb1 ass*
⋀*ass val*. *val-ifex x ass* = *val-ifex yb2 ass* **by** *blast*+
from *this IF*(*1*,*2*) *x-def 0*(*5*) *y'-def 0*(*6*) **have** *x* = *yb1 x* = *yb2* **by** *fastforce*+
from *this* **have** *yb1* = *yb2* **by** *auto*
from *0*(*6*) *y'-def this* **have** *False* **by** *auto*
thus *?thesis* **..**
qed
qed
qed (*fastforce intro*: *roifex-Falseif-unique roifex-Trueif-unique*)+

qed (*fastforce intro*: *roifex-Falseif-unique*[*symmetric*] *roifex-Trueif-unique*[*symmetric*])+

theorem *bf-ifex-rel-single*: *single-valued bf-ifex-rel single-valued* (*bf-ifex-rel*$^{-1}$)
unfolding *single-valued-def bf-ifex-rel-def* **using** *ro-ifex-unique* **by** *auto*

lemma *bf-ifex-eq*: (*af*, *at*) ∈ *bf-ifex-rel* ⟹ (*bf*, *bt*) ∈ *bf-ifex-rel* ⟹ (*af* = *bf*) ⟷ (*at* = *bt*)
unfolding *bf-ifex-rel-def* **using** *ro-ifex-unique* **by** *auto*

lemma *nonempty-if-var-set*: *ifex-vars* (*IF v t e*) ≠ [] **by** *auto*

fun *restrict* **where**
restrict (*IF v t e*) *var val* = (*let rt* = *restrict t var val*; *re* = *restrict e var val in*
(*if v* = *var then* (*if val then rt else re*) *else* (*IF v rt re*))) |
restrict i - - = *i*

declare *Let-def*[*simp*]

lemma *not-element-restrict*: *var* ∉ *ifex-var-set* (*restrict b var val*)
by (*induction b*) *auto*

lemma *restrict-assignment*: *val-ifex b* (*ass*(*var* := *val*)) ⟷ *val-ifex* (*restrict b var val*) *ass*
by (*induction b*) *auto*

lemma *restrict-variables-subset*: *ifex-var-set* (*restrict b var val*) ⊆ *ifex-var-set b*
by (*induction b*) *auto*

lemma *restrict-ifex-ordered-invar*: *ifex-ordered b* ⟹ *ifex-ordered* (*restrict b var val*)
using *restrict-variables-subset* **by** (*induction b*) (*fastforce*)+

lemma *restrict-val-invar*: ∀ *ass*. *a ass* = *val-ifex b ass* ⟹
(*bf-restrict var val a*) *ass* = *val-ifex* (*restrict b var val*) *ass*
unfolding *bf-restrict-def* **using** *restrict-assignment* **by** *simp*

lemma *restrict-untouched-id*: *x* ∉ *ifex-var-set t* ⟹ *restrict t x val* = *t*
proof(*induction t*)
case (*IF v t e*)
from *IF.prems* **have** *x* ∉ *ifex-var-set t x* ∉ *ifex-var-set e* **by** *simp-all*
note *mIH* = *IF.IH*(*1*)[*OF this*(*1*)] *IF.IH*(*2*)[*OF this*(*2*)]
from *IF.prems* **have** *x* ≠ *v* **by** *simp*
thus *?case* **unfolding** *restrict.simps Let-def mIH* **by** *simp*
qed *simp-all*

fun *ifex-top-var* :: *'a ifex* ⇒ *'a option* **where**
ifex-top-var (*IF v t e*) = *Some v* |
ifex-top-var - = *None*

fun *restrict-top* :: (*'a* :: *linorder*) *ifex* ⇒ *'a* ⇒ *bool* ⇒ *'a ifex* **where**
restrict-top (*IF v t e*) *var val* = (*if v* = *var then* (*if val then t else e*) *else* (*IF v t e*)) |
restrict-top i - - = *i*

lemma *restrict-top-id*: *ifex-ordered e* ⟹ *ifex-top-var e* = *Some v* ⟹ *v′* < *v* ⟹ *restrict-top e v′ val* = *e*
by(*induction e*) *auto*

lemma *restrict-id*: *ifex-ordered e* ⟹ *ifex-top-var e* = *Some v* ⟹ *v′* < *v* ⟹ *restrict e v′ val* = *e*
proof(*induction e arbitrary*: *v*)
case (*IF w e1 e2*) **thus** *?case* **by** (*cases e1*; *cases e2*; *force*)
qed(*auto*)

lemma *restrict-top-IF-id*: *ifex-ordered* (*IF v t e*) ⟹ *v′* < *v* ⟹ *restrict-top* (*IF v t e*) *v′ val* = (*IF v t e*)
using *restrict-top-id* **by** *auto*

lemma *restrict-IF-id*: **assumes** *o*: *ifex-ordered* (*IF v t e*) **assumes** *le*: *v′* < *v*
shows *restrict* (*IF v t e*) *v′ val* = (*IF v t e*)
using *restrict-id*[*OF o*, *unfolded ifex-top-var.simps*, *OF refl le*, *of val*] **.**

lemma *restrict-top-eq*: *ifex-ordered* (*IF v t e*) ⟹ *restrict* (*IF v t e*) *v val* = *restrict-top* (*IF v t e*) *v val*
using *restrict-untouched-id* **by** *auto*

lemma *restrict-top-ifex-ordered-invar*: *ifex-ordered b* ⟹ *ifex-ordered* (*restrict-top b var val*)
by (*induction b*) *simp-all*

fun *lowest-tops* :: (*'a* :: *linorder*) *ifex list* ⇒ *'a option* **where**
lowest-tops [] = *None* |
lowest-tops ((*IF v - -*)#*r*) = *Some* (*case lowest-tops r of Some u* ⇒ (*min u v*) | *None* ⇒ *v*) |
lowest-tops (-#*r*) = *lowest-tops r*

lemma *lowest-tops-NoneD*: *lowest-tops k* = *None* ⟹ (¬(∃ *v t e*. ((*IF v t e*) ∈ *set k*)))
by (*induction k rule*: *lowest-tops.induct*) *simp-all*

lemma *lowest-tops-in*: *lowest-tops k* = *Some l* ⟹ *l* ∈ *set* (*concat* (*map ifex-vars k*))
by(*induction k rule*: *lowest-tops.induct*) (*simp-all split*: *option.splits if-splits add*: *min-def*)

definition *IFC v t e* ≡ (*if t* = *e then t else IF v t e*)

```
function ifex-ite :: 'a ifex ⇒ 'a ifex ⇒ 'a ifex ⇒ ('a :: linorder) ifex where
  ifex-ite i t e = (case lowest-tops [i, t, e] of Some x ⇒
                         (IFC x (ifex-ite (restrict-top i x True) (restrict-top t x True)
(restrict-top e x True))
                                 (ifex-ite (restrict-top i x False) (restrict-top t x False)
(restrict-top e x False)))
                    | None ⇒ (case i of Trueif ⇒ t | Falseif ⇒ e))
by pat-completeness auto

lemma restrict-size-le: size (restrict-top k var val) ≤ size k
  by (induction k, auto)

lemma restrict-size-less: ifex-top-var k = Some var ⟹ size (restrict-top k var
val) < size k
  by (induction k, auto)

lemma lowest-tops-cases:
lowest-tops [i, t, e] = Some var ⟹ ifex-top-var i = Some var ∨ ifex-top-var t
                                  = Some var ∨ ifex-top-var e = Some var
  by ((cases i; cases t; cases e), auto simp add: min-def)

lemma lowest-tops-lowest: lowest-tops es = Some a ⟹ e ∈ set es ⟹ ifex-ordered
e ⟹ v ∈ ifex-var-set e ⟹ a ≤ v
proof (induction arbitrary: a rule: lowest-tops.induct)
  case 2 thus ?case
  proof(cases e)
    case IF with 2 show ?thesis
     apply (simp add: min-def Ball-def less-imp-le split: if-splits option.splits)
       apply (meson less-imp-le lowest-tops-NoneD order-refl)
      by fastforce+
  qed simp+
qed fastforce+

lemma termlemma2: lowest-tops [i, t, e] = Some xa ⟹
  (size (restrict-top i xa val) + size (restrict-top t xa val) + size (restrict-top e xa
val)) <
  (size i + size t + size e)
  using restrict-size-le[of i xa val] restrict-size-le[of t xa val]  restrict-size-le[of e xa
val]
  by (auto dest!: lowest-tops-cases restrict-size-less[of - - val])

lemma termlemma: lowest-tops [i, t, e] = Some xa ⟹
       (case (restrict-top i xa val, restrict-top t xa val, restrict-top e xa val) of
             (i, t, e) ⇒ size i + size t + size e) <
       (case (i, t, e) of (i, t, e) ⇒ size i + size t + size e)
  using termlemma2 by fast

termination ifex-ite
  by (relation measure (λ(i,t,e). size i + size t + size e), rule wf-measure, unfold
```

in-measure)
  (*simp-all only*: *termlemma*)

definition *const x - = x*
declare *const-def*[*simp*]
lemma *rel-true-false*: (*a*, *Trueif*) ∈ *bf-ifex-rel* ⟹ *a* = *const True* (*a*, *Falseif*) ∈ *bf-ifex-rel* ⟹ *a* = *const False*
  unfolding *fun-eq-iff const-def*
  unfolding *bf-ifex-rel-def*
  by *simp-all*

lemma *rel-if*: (*a*, *IF v t e*) ∈ *bf-ifex-rel* ⟹ (*ta*, *t*) ∈ *bf-ifex-rel* ⟹ (*ea*, *e*) ∈ *bf-ifex-rel* ⟹ *a* = (*λas. if as v then ta as else ea as*)
  unfolding *fun-eq-iff const-def*
  unfolding *bf-ifex-rel-def*
  by *simp-all*

lemma *ifex-ordered-implied*: (*a*, *b*) ∈ *bf-ifex-rel* ⟹ *ifex-ordered b* **unfolding** *bf-ifex-rel-def* **by** *simp*
lemma *ifex-minimal-implied*: (*a*, *b*) ∈ *bf-ifex-rel* ⟹ *ifex-minimal b* **unfolding** *bf-ifex-rel-def* **by** *simp*

lemma *ifex-ite-induct2*[*case-names Trueif Falseif IF*]:
  (⋀*i t e. lowest-tops* [*i*, *t*, *e*] = *None* ⟹ *i* = *Trueif* ⟹ *sentence i t e*) ⟹
  (⋀*i t e. lowest-tops* [*i*, *t*, *e*] = *None* ⟹ *i* = *Falseif* ⟹ *sentence i t e*) ⟹
  (⋀*i t e a. sentence* (*restrict-top i a True*) (*restrict-top t a True*) (*restrict-top e a True*) ⟹
        sentence (*restrict-top i a False*) (*restrict-top t a False*) (*restrict-top e a False*) ⟹
   lowest-tops [*i*, *t*, *e*] = *Some a* ⟹ *sentence i t e*) ⟹ *sentence i t e*
proof(*induction i t e rule*: *ifex-ite.induct*, *goal-cases*)
  case (*1 i t e*) **show** *?case*
  proof(*cases lowest-tops* [*i*, *t*, *e*])
    case *None* **thus** *?thesis* **by** (*cases i*) (*auto intro*: *1*)
  next
    case (*Some a*) **thus** *?thesis* **by**(*auto intro*: *1*)
  qed
qed

lemma *ifex-ite-induct*[*case-names Trueif Falseif IF*]:
  (⋀*i t e. lowest-tops* [*i*, *t*, *e*] = *None* ⟹ *i* = *Trueif* ⟹ *P i t e*) ⟹
  (⋀*i t e. lowest-tops* [*i*, *t*, *e*] = *None* ⟹ *i* = *Falseif* ⟹ *P i t e*) ⟹
  (⋀*i t e a.* (⋀*val. P* (*restrict-top i a val*) (*restrict-top t a val*) (*restrict-top e a val*)) ⟹
   lowest-tops [*i*, *t*, *e*] = *Some a* ⟹ *P i t e*) ⟹ *P i t e*
proof(*induction i t e rule*: *ifex-ite-induct2*)

case (*IF i t e a*)
have $\bigwedge$*val*. (*P* (*restrict-top i a val*) (*restrict-top t a val*) (*restrict-top e a val*))
by (*case-tac val*) (*clarsimp*, *blast intro*: *IF*)+
with *IF* **show** *?case* **by** *blast*
qed *blast*+

lemma *restrict-top-subset*: *x* ∈ *ifex-var-set* (*restrict-top i vr vl*) ⟹ *x* ∈ *ifex-var-set* *i*
by(*induction i*) (*simp-all split*: *if-splits*)

lemma *ifex-vars-subset*: *x* ∈ *ifex-var-set* (*ifex-ite i t e*) ⟹ (*x* ∈ *ifex-var-set i*) ∨ (*x* ∈ *ifex-var-set t*) ∨ (*x* ∈ *ifex-var-set e*)
proof(*induction rule*: *ifex-ite-induct2*)
case (*IF i t e a*)
have *x* ∈ {*x*. *x* = *a*} ∨ *x* ∈ (*ifex-var-set* (*ifex-ite* (*restrict-top i a True*) (*restrict-top t a True*) (*restrict-top e a True*))) ∨ *x* ∈ (*ifex-var-set* (*ifex-ite* (*restrict-top i a False*) (*restrict-top t a False*) (*restrict-top e a False*)))
using *IF* **by**(*simp add*: *IFC-def split*: *if-splits*)
hence *x* = *a* ∨
x ∈ (*ifex-var-set* (*restrict-top i a True*)) ∨ *x* ∈ (*ifex-var-set* (*restrict-top t a True*)) ∨ *x* ∈ (*ifex-var-set* (*restrict-top e a True*)) ∨
x ∈ (*ifex-var-set* (*restrict-top i a False*)) ∨ *x* ∈ (*ifex-var-set* (*restrict-top t a False*)) ∨ *x* ∈ (*ifex-var-set* (*restrict-top e a False*))
using *IF* **by** *blast*
thus *?case*
using *restrict-top-subset* **apply** −
apply(*erule disjE*)
subgoal using *lowest-tops-in*[*OF IF*(*3*)] **apply**(*simp only*: *set-concat set-map set-simps*) **by** *blast*
by *blast*
qed *simp-all*

lemma *three-ins*: *i* ∈ *set* [*i*, *t*, *e*] *t* ∈ *set* [*i*, *t*, *e*] *e* ∈ *set* [*i*, *t*, *e*] **by** *simp-all*

lemma *hlp3*: *lowest-tops* (*IF v uu uv* # *r*) ≠ *lowest-tops r* ⟹ *lowest-tops* (*IF v uu uv* # *r*) = *Some v*
by(*simp add*: *min-def split*: *option.splits if-splits*)

lemma *hlp2*: *IF vi vt ve* ∈ *set is* ⟹ *lowest-tops is* = *Some a* ⟹ *a* ≤ *vi*
apply(*induction is arbitrary*: *vt ve a rule*: *lowest-tops.induct*)
prefer *2*
subgoal
apply(*auto simp add*: *min-def split*: *if-splits option.splits dest*: *lowest-tops-NoneD*)
by (*meson le-cases order-trans*)
by (*auto*)

lemma *hlp1*: *i* ∈ *set is* ⟹ *lowest-tops is* = *Some a* ⟹ *ifex-ordered i* ⟹ *a* ∉ (*ifex-var-set* (*restrict-top i a val*))
proof(*rule ccontr*, *unfold not-not*, *goal-cases*)

case *1*
from *1(4)* **obtain** *vi vt ve* **where** *vi*: *i* = *IF vi vt ve* **by**(*cases i*) *simp-all*
with *1* **have** *ne*: *vi* ≠ *a* **by**(*simp split*: *if-splits*) *blast+*
moreover have *vi* ≤ *a* **using** *1(3,4)* **proof**(−,*goal-cases*)
case *1*
hence *a* ∈ (*ifex-var-set vt*) ∨ *a* ∈ (*ifex-var-set ve*) **using** *ne* **by**(*simp add*: *vi*)
thus *?case* **using** ‹*ifex-ordered i*› *vi* **using** *less-imp-le* **by** *auto*
qed
moreover have *a* ≤ *vi* **using** *1(1)* **unfolding** *vi* **using** *1(2) hlp2* **by** *metis*
ultimately show *False* **by** *simp*
qed

lemma *order-ifex-ite-invar*: *ifex-ordered i* ⟹ *ifex-ordered t* ⟹ *ifex-ordered e* ⟹ *ifex-ordered* (*ifex-ite i t e*)
proof(*induction i t e rule*: *ifex-ite-induct*)
case (*IF i t e*) **note** *goal1* = *IF*
note *l* = *restrict-top-ifex-ordered-invar*
note *l*[*OF goal1(3)*] *l*[*OF goal1(4)*] *l*[*OF goal1(5)*]
note *mIH* = *goal1(1)*[*OF this*]
note *blubb* = *lowest-tops-lowest*[*OF goal1(2) - - restrict-top-subset*]
show *?case* **using** *mIH*
by (*subst ifex-ite.simps*,
auto simp del: *ifex-ite.simps*
simp add: *IFC-def goal1(2) hlp1*[*OF three-ins(1) goal1(2) goal1(3)*] *hlp1*[*OF three-ins(2) goal1(2) goal1(4)*] *hlp1*[*OF three-ins(3) goal1(2) goal1(5)*]
dest: *ifex-vars-subset blubb*[*OF three-ins(1) goal1(3)*] *blubb*[*OF three-ins(2) goal1(4)*] *blubb*[*OF three-ins(3) goal1(5)*]
intro!: *le-neq-trans*)
qed *simp-all*

lemma *ifc-split*: *P* (*IFC v t e*) ⟷ ((*t* = *e*) ⟶ *P t*) ∧ (*t* ≠ *e* ⟶ *P* (*IF v t e*))
unfolding *IFC-def* **by** *simp*

lemma *restrict-top-ifex-minimal-invar*: *ifex-minimal i* ⟹ *ifex-minimal* (*restrict-top i a val*)
by(*induction i*) *simp-all*

lemma *minimal-ifex-ite-invar*: *ifex-minimal i* ⟹ *ifex-minimal t* ⟹ *ifex-minimal e* ⟹ *ifex-minimal* (*ifex-ite i t e*)
by(*induction i t e rule*: *ifex-ite-induct*) (*simp-all split*: *ifc-split option.split add*: *restrict-top-ifex-minimal-invar*)

lemma *restrict-top-bf*: *i* ∈ *set is* ⟹ *lowest-tops is* = *Some vr* ⟹
ifex-ordered i ⟹ (⋀*ass*. *fi ass* = *val-ifex i ass*) ⟹ *val-ifex* (*restrict-top i vr vl*) *ass* = *bf-restrict vr vl fi ass*
proof(*cases i*, *goal-cases*)
case (*3 x31 x32 x33*) **note** *goal3* = *3*
have *rr*: *restrict-top i vr vl* = *restrict i vr vl*
proof(*cases x31* = *vr*)

case *True*
note *uf* = *restrict-top-eq*[*OF goal3*(*3*)[*unfolded goal3*(*5*)], *symmetric*, *unfolded goal3*(*5*)[*symmetric*], *unfolded True*]
thus *?thesis* **.**
next
case *False*
have *1*: *restrict-top i vr vl* = *i* **by** (*simp add*: *False goal3*(*5*))
have *vr* < *x31* **using** *le-neq-trans*[*OF hlp2*[*OF goal3*(*1*)[*unfolded goal3*(*5*)] *goal3*(*2*)] *False*[*symmetric*]] **by** *blast*
with *goal3*(*3*,*5*) **have** *2*: *restrict i vr vl* = *i* **using** *restrict-IF-id* **by** *blast*
show *?thesis* **unfolding** *1 2* **..**
qed
show *?case* **unfolding** *rr* **by**(*simp add*: *goal3*(*4*) *restrict-val-invar*[*symmetric*])
qed (*simp-all add*: *bf-restrict-def*)

lemma *val-ifex-ite*:
(⋀*ass. fi ass* = *val-ifex i ass*) ⟹
(⋀*ass. ft ass* = *val-ifex t ass*) ⟹
(⋀*ass. fe ass* = *val-ifex e ass*) ⟹
ifex-ordered i ⟹ *ifex-ordered t* ⟹ *ifex-ordered e* ⟹
(*bf-ite fi ft fe*) *ass* = *val-ifex* (*ifex-ite i t e*) *ass*
proof(*induction i t e arbitrary*: *fi ft fe rule*: *ifex-ite-induct*)
case (*IF i t e a*)
note *mIH* = *IF*(*1*)[*OF refl refl refl*
restrict-top-ifex-ordered-invar[*OF IF*(*6*)]
restrict-top-ifex-ordered-invar[*OF IF*(*7*)]
restrict-top-ifex-ordered-invar[*OF IF*(*8*)], *symmetric*]
note *uf1* = *restrict-top-bf*[*OF three-ins*(*1*) *IF*(*2*) ‹*ifex-ordered i*› *IF*(*3*)]
restrict-top-bf[*OF three-ins*(*2*) *IF*(*2*) ‹*ifex-ordered t*› *IF*(*4*)]
restrict-top-bf[*OF three-ins*(*3*) *IF*(*2*) ‹*ifex-ordered e*› *IF*(*5*)]
show *?case*
by(*rule trans*[*OF brace90shannon*[**where** *i*=*a*]])
(*auto simp*: *restrict-top-ifex-ordered-invar IF*(*1*,*2*,*6*−*8*) *uf1 mIH bf-ite-def*[*of* λ*l. l a*]
split: *ifc-split*)
qed (*simp add*: *bf-ite-def bf-ifex-rel-def*)+

theorem *ifex-ite-rel-bf*:
(*fi*,*i*) ∈ *bf-ifex-rel* ⟹
(*ft*,*t*) ∈ *bf-ifex-rel* ⟹
(*fe*,*e*) ∈ *bf-ifex-rel* ⟹
((*bf-ite fi ft fe*), (*ifex-ite i t e*)) ∈ *bf-ifex-rel*
by (*auto simp add*: *bf-ifex-rel-def order-ifex-ite-invar minimal-ifex-ite-invar val-ifex-ite*
simp del: *ifex-ite.simps*)

definition *param-opt* **where** *param-opt i t e* =
(*if i* = *Trueif then Some t else*
if i = *Falseif then Some e else*
if t = *Trueif* ∧ *e* = *Falseif then Some i else*

```
  if t = e then Some t else
  if e = Trueif ∧ i = t then Some Trueif else
  if t = Falseif ∧ i = e then Some Falseif else
  None)

lemma param-opt-ifex-ite-eq: ro-ifex i ⟹ ro-ifex t ⟹ ro-ifex e ⟹
    param-opt i t e = Some r ⟹ r = ifex-ite i t e
 apply(rule ro-ifex-unique)
  subgoal by (subst (asm) param-opt-def) (simp split: if-split-asm)
  subgoal using order-ifex-ite-invar minimal-ifex-ite-invar by (blast)
  by (subst val-ifex-ite[symmetric])
   (auto split: if-split-asm simp add: bf-ite-def param-opt-def val-ifex-ite[symmetric])


function ifex-ite-opt :: 'a ifex ⇒ 'a ifex ⇒ 'a ifex ⇒ ('a :: linorder) ifex where
 ifex-ite-opt i t e = (case param-opt i t e of Some b ⇒ b | None ⇒
                    (case lowest-tops [i, t, e] of Some x ⇒
                   (IFC x (ifex-ite-opt (restrict-top i x True) (restrict-top t x True)
                                (restrict-top e x True))
                       (ifex-ite-opt (restrict-top i x False) (restrict-top t x False)
                                (restrict-top e x False)))
                  | None ⇒ (case i of Trueif ⇒ t | Falseif ⇒ e)))
by pat-completeness auto

termination ifex-ite-opt
 by (relation measure (λ(i,t,e). size i + size t + size e), rule wf-measure, unfold
in-measure)
    (simp-all only: termlemma)

lemma ifex-ite-opt-eq:
 ro-ifex i ⟹ ro-ifex t ⟹ ro-ifex e ⟹ ifex-ite-opt i t e = ifex-ite i t e
 apply(induction i t e rule: ifex-ite-opt.induct)
 apply(subst ifex-ite-opt.simps)
 apply(rename-tac i t e)
 apply(case-tac ∃ r. param-opt i t e = Some r)
  subgoal
   apply(simp del: ifex-ite.simps restrict-top.simps lowest-tops.simps)
   apply(rule param-opt-ifex-ite-eq)
   by (auto simp add: bf-ifex-rel-def)
  subgoal for i t e
   apply(clarsimp simp del: restrict-top.simps ifex-ite.simps ifex-ite-opt.simps)
   apply(cases lowest-tops [i,t,e] = None)
    subgoal by clarsimp
    subgoal
     apply(clarsimp simp del: restrict-top.simps ifex-ite.simps ifex-ite-opt.simps)
     apply(subst ifex-ite.simps)
     apply(rename-tac y)
      apply(subgoal-tac (ifex-ite-opt (restrict-top i y True) (restrict-top t y True)
(restrict-top e y True)) =
```

(*ifex-ite* (*restrict-top* *i* *y* *True*) (*restrict-top* *t* *y* *True*) (*restrict-top* *e* *y* *True*)))
apply(*subgoal-tac* (*ifex-ite-opt* (*restrict-top* *i* *y* *False*) (*restrict-top* *t* *y* *False*) (*restrict-top* *e* *y* *False*)) =
(*ifex-ite* (*restrict-top* *i* *y* *False*) (*restrict-top* *t* *y* *False*) (*restrict-top* *e* *y* *False*)))
subgoal by *force*
subgoal using *restrict-top-ifex-minimal-invar restrict-top-ifex-ordered-invar* **by** *metis*
subgoal using *restrict-top-ifex-minimal-invar restrict-top-ifex-ordered-invar* **by** *metis*
done
done
done

lemma *ro-ifexI*: (*a*,*b*) ∈ *bf-ifex-rel* ⟹ *ro-ifex* *b* **by** (*simp add*: *ifex-minimal-implied* *ifex-ordered-implied*)

theorem *ifex-ite-opt-rel-bf*:
(*fi*,*i*) ∈ *bf-ifex-rel* ⟹
(*ft*,*t*) ∈ *bf-ifex-rel* ⟹
(*fe*,*e*) ∈ *bf-ifex-rel* ⟹
((*bf-ite* *fi* *ft* *fe*), (*ifex-ite-opt* *i* *t* *e*)) ∈ *bf-ifex-rel*
using *ifex-ite-rel-bf ifex-ite-opt-eq ro-ifexI* **by** *metis*

lemma *restrict-top-bf-ifex-rel*:
(*f*, *i*) ∈ *bf-ifex-rel* ⟹ ∃ *f'*. (*f'*, *restrict-top* *i* *var* *val*) ∈ *bf-ifex-rel*
unfolding *bf-ifex-rel-def* **using** *restrict-top-ifex-minimal-invar restrict-top-ifex-ordered-invar* **by** *fast*

lemma *param-opt-lowest-tops-lem*: *param-opt* *i* *t* *e* = *None* ⟹ ∃ *y*. *lowest-tops* [*i*,*t*,*e*] = *Some* *y*
by (*cases* *i*) (*auto simp add*: *param-opt-def*)

fun *ifex-sat* **where**
ifex-sat Trueif = *Some* (*const False*) |
ifex-sat Falseif = *None* |
ifex-sat (*IF* *v* *t* *e*) =
(*case ifex-sat e of*
Some a ⇒ *Some* (*a*(*v*:=*False*)) |
None ⇒ (*case ifex-sat t of*
Some a ⇒ *Some* (*a*(*v*:=*True*)) |
None ⇒ *None*))

lemma *ifex-sat-untouched-False*: *v* ∉ *ifex-var-set* *i* ⟹ *ifex-sat* *i* = *Some* *a* ⟹ *a* *v* = *False*

proof(*induction i arbitrary*: *a*)
case (*IF v1 t e*)
have *ni*: *v* ∉ *ifex-var-set t v* ∉ *ifex-var-set e* **using** *IF.prems*(*1*) **by** *simp-all*
have *ne*: *v1* ≠ *v* **using** *IF.prems*(*1*) **by** *force*
show *?case* **proof**(*cases ifex-sat e*)
case (*Some as*)
with *IF.prems*(*2*) **have** *au*: *a* = *as*(*v1* := *False*) **by** *simp*
moreover from *IF.IH*(*2*)[*OF ni*(*2*)] **have** *as v* = *False* **using** *Some* **.**
ultimately show *?thesis* **using** *ne* **by** *simp*
next
case *None*
obtain *as* **where** *Some*: *ifex-sat t* = *Some as* **using** *None IF.prems*(*2*) **by** *fastforce*
with *IF.prems*(*2*) *None* **have** *au*: *a* = *as*(*v1* := *True*) **by**(*simp*)
moreover from *IF.IH*(*1*)[*OF ni*(*1*)] **have** *as v* = *False* **using** *Some* **.**
ultimately show *?thesis* **using** *ne* **by** *simp*
qed
qed(*simp-all add*: *fun-eq-iff*)

lemma *ifex-upd-other*: *v* ∉ *ifex-var-set i* ⟹ *val-ifex i* (*a*(*v*:=*any*)) = *val-ifex i a*
proof(*induction i*)
case (*IF v1 t e*)
have *prems*: *v* ∉ *ifex-var-set t v* ∉ *ifex-var-set e* **using** *IF.prems* **by** *simp-all*
from *IF.prems* **have** *ne*: *v1* ≠ *v* **by** *clarsimp*
show *?case* **by**(*simp only*: *val-ifex.simps fun-upd-other*[*OF ne*] *ifex-vars.simps IF.IH*(*1*)[*OF prems*(*1*)] *IF.IH*(*2*)[*OF prems*(*2*)] *split*: *if-splits*)
qed *simp-all*

fun *ifex-no-twice* **where**
ifex-no-twice (*IF v t e*) = (
v ∉ (*ifex-var-set t* ∪ *ifex-var-set e*) ∧
ifex-no-twice t ∧ *ifex-no-twice e*) |
ifex-no-twice - = *True*
lemma *ordered-ifex-no-twiceI*: *ifex-ordered i* ⟹ *ifex-no-twice i*
by(*induction i*) (*simp-all*,*blast*)

lemma *ifex-sat-NoneD*: *ifex-sat i* = *None* ⟹ *val-ifex i ass* = *False*
by(*induction i*) (*simp-all split*: *option.splits*)
lemma *ifex-sat-SomeD*: *ifex-no-twice i* ⟹ *ifex-sat i* = *Some ass* ⟹ *val-ifex i ass* = *True*
proof(*induction i arbitrary*: *ass*)
case (*IF v t e*)
have *ni*: *v* ∉ *ifex-var-set t v* ∉ *ifex-var-set e* **using** *IF.prems*(*1*) **by** *simp-all*
note *IF.prems*[*unfolded ifex-sat.simps*]
thus *?case* **proof**(*cases ifex-sat e*)
case (*Some a*) **thus** *?thesis* **using** *IF.prems*
apply(*clarsimp simp only*: *val-ifex.simps ifex-sat.simps option.simps fun-upd-same if-False ifex-upd-other*[*OF ni*(*2*)])

apply(*rule IF.IH*(*2*), *simp-all*)
done
next
case *None*
obtain *a* **where** *Some*: *ifex-sat t* = *Some a* **using** *None IF.prems*(*2*) **by** *fastforce*
thus *?thesis* **using** *IF.prems*
by(*clarsimp simp only*: *val-ifex.simps ifex-sat.simps option.simps fun-upd-same if-True None ifex-upd-other*[*OF ni*(*1*)])
(*rule IF.IH*(*1*), *simp-all*)
qed
qed *simp-all*
lemma *ifex-sat-NoneI*: *ifex-no-twice i* ⟹ (⋀*ass. val-ifex i ass* = *False*) ⟹ *ifex-sat i* = *None*

proof(*rule ccontr*, *goal-cases*)
case *1*
from *1*(*3*) **obtain** *as* **where** *ifex-sat i* = *Some as* **by** *blast*
from *ifex-sat-SomeD*[*OF 1*(*1*) *this*] **show** *False* **using** *1*(*2*) **by** *simp*
qed

fun *ifex-sat-list* **where**
ifex-sat-list Trueif = *Some* [] |
ifex-sat-list Falseif = *None* |
ifex-sat-list (*IF v t e*) =
(*case ifex-sat-list e of*
Some a ⇒ *Some* ((*v*,*False*)#*a*) |
None ⇒ (*case ifex-sat-list t of*
Some a ⇒ *Some* ((*v*,*True*)#*a*) |
None ⇒ *None*))

definition *update-assignment-alt u as* = (*λv. case map-of u v of None* ⇒ *as v* | *Some n* ⇒ *n*)
fun *update-assignment* **where**
update-assignment ((*v*,*u*)#*us*) *as* = (*update-assignment us as*)(*v*:=*u*) |
update-assignment [] *as* = *as*

lemma *update-assignment-notin*: *a* ∉ *fst* ‘ *set us* ⟹ *update-assignment us as a* = *as a*
by(*induction us*) *clarsimp*+

lemma *update-assignment-alt*: *update-assignment u as* = *update-assignment-alt u as*
by(*induction u arbitrary*: *as*) (*clarsimp simp*: *update-assignment-alt-def fun-eq-iff*)+

lemma *update-assignment*: *distinct* (*map fst* ((*v*,*u*)#*us*)) ⟹ *update-assignment* ((*v*,*u*)#*us*) *as* = *update-assignment us* (*as*(*v*:=*u*))
unfolding *update-assignment-alt update-assignment-alt-def*

unfolding *fun-eq-iff*
by(*clarsimp split*: *option.splits*) *force*

lemma *ass-upd-same*: *update-assignment* ((*v*, *u*) # *a*) *ass v* = *u* **by** *simp*

lemma *ifex-sat-list-subset*: *ifex-sat-list t* = *Some u* ⟹ *fst* ‘ *set u* ⊆ *ifex-var-set t*
proof(*induction t arbitrary*: *u*)
  case (*IF v t e*)
  show *?case*
  proof(*cases ifex-sat-list e*)
    case (*Some ue*)
    note *IF.IH*(*2*)[*OF this*]
    hence *fst* ‘ *set ue* ⊆ *ifex-var-set* (*IF v t e*) **by** *simp blast*
    moreover have *fst* ‘ *set u* = *insert v* (*fst* ‘ *set ue*) **using** *IF.prems Some* **by** *force*
    ultimately show *?thesis* **by** *simp*
  next
    case *None*
    with *IF.prems* **obtain** *ut* **where** *Some*: *ifex-sat-list t* = *Some ut* **by**(*simp split*: *option.splits*)
    note *IF.IH*(*1*)[*OF this*]
    hence *fst* ‘ *set ut* ⊆ *ifex-var-set* (*IF v t e*) **by** *simp blast*
    moreover have *fst* ‘ *set u* = *insert v* (*fst* ‘ *set ut*) **using** *IF.prems None Some* **by** *force*
    ultimately show *?thesis* **by** *simp*
  qed
qed *simp-all*

lemma *sat-list-distinct*: *ifex-no-twice t* ⟹ *ifex-sat-list t* = *Some u* ⟹ *distinct* (*map fst u*)
proof(*induction t arbitrary*: *u*)
  case (*IF v t e*)
  from *IF.prems* **have** *nt*: *ifex-no-twice t ifex-no-twice e* **by** *simp-all*
  note *mIH* = *IF.IH*(*1*)[*OF this*(*1*)] *IF.IH*(*2*)[*OF this*(*2*)]
  show *?case*
  proof(*cases ifex-sat-list e*)
    case (*Some a*)
    note *mIH* = *mIH*(*2*)[*OF this*]
    thus *?thesis* **using** *IF.prems ifex-sat-list.simps Some ifex-sat-list-subset* **by** *fastforce*
  next
    case *None*
    with *IF.prems* **obtain** *ut* **where** *Some*: *ifex-sat-list t* = *Some ut* **by**(*simp split*: *option.splits*)
    note *mIH*(*1*)[*OF this*]
    thus *?thesis* **using** *IF.prems ifex-sat-list.simps None Some ifex-sat-list-subset* **by** *fastforce*
  qed

qed *simp-all*

lemma *ifex-sat-list-NoneD*: *ifex-sat-list i = None* $\Longrightarrow$ *val-ifex i ass = False*
by(*induction i*) (*simp-all split*: *option.splits*)
lemma *ifex-sat-list-SomeD*: *ifex-no-twice i* $\Longrightarrow$ *ifex-sat-list i = Some u* $\Longrightarrow$ *ass = update-assignment u ass'* $\Longrightarrow$ *val-ifex i ass = True*
proof(*induction i arbitrary*: *ass ass' u*)
case (*IF v t e*)
have *nt*: *ifex-no-twice t ifex-no-twice e* **using** *IF.prems*(*1*) **by** *simp-all*
have *ni*: *v* ∉ *ifex-var-set t v* ∉ *ifex-var-set e* **using** *IF.prems*(*1*) **by** *simp-all*
note *IF.prems*[*unfolded ifex-sat.simps*]
thus *?case* **proof**(*cases ifex-sat-list e*)
case (*Some a*)
have *ef*: *u* = (*v*, *False*) # *a* **using** *IF.prems*(*2*) *Some* **by** *simp*
from *IF.prems*(*3*) **have** *au*: *ass = update-assignment a* (*ass'*(*v* := *False*)) **unfolding** *ef* **using** *update-assignment*[*OF sat-list-distinct*[*OF IF.prems*(*1*,*2*), *unfolded ef*]] **by** *presburger*
have *avF*: *ass v = False* **using** *IF.prems*(*3*)[*symmetric*] **unfolding** *ef* **by** *clarsimp*
show *?thesis* **using** *IF.IH*(*2*)[*OF nt*(*2*) *Some au*] *Some IF.prems*(*2*) *avF* **by** *simp*
next
case *None*
obtain *a* **where** *Some*: *ifex-sat-list t = Some a* **using** *None IF.prems*(*2*) **by** *fastforce*
have *ef*: *u* = (*v*, *True*) # *a* **using** *IF.prems*(*2*) *None Some* **by** *simp*
from *IF.prems*(*3*) **have** *au*: *ass = update-assignment a* (*ass'*(*v* := *True*)) **unfolding** *ef* **using** *update-assignment*[*OF sat-list-distinct*[*OF IF.prems*(*1*,*2*), *unfolded ef*]] **by** *presburger*
have *avT*: *ass v = True* **using** *IF.prems*(*3*)[*symmetric*] **unfolding** *ef* **by** *clarsimp*
show *?thesis* **using** *IF.IH*(*1*)[*OF nt*(*1*) *Some au*] *Some IF.prems*(*2*) *avT* **by** *simp*
qed
qed *simp-all*

fun *sat-list-to-bdt* **where**
sat-list-to-bdt [] = *Trueif* |
sat-list-to-bdt ((*v*,*u*)#*us*) = (*if u then IF v* (*sat-list-to-bdt us*) *Falseif else IF v Falseif* (*sat-list-to-bdt us*))

lemma *ifex-sat-list i = Some u* $\Longrightarrow$ *val-ifex* (*sat-list-to-bdt u*) *as* $\Longrightarrow$ *val-ifex i as*
proof(*induction i arbitrary*: *u*)
case (*IF v t e*)
show *?case* **proof**(*cases ifex-sat-list e*)
case (*Some a*)
note *mIH = IF.IH*(*2*)[*OF this*]
have *ef*: *u* = (*v*, *False*) # *a* **using** *IF.prems*(*1*) *Some* **by** *simp*
have *avF*: *as v = False* **using** *IF.prems*(*2*) **unfolding** *ef* **by**(*simp split*: *if-splits*)

have *val-ifex* (*sat-list-to-bdt a*) *as* **using** *IF.prems*(*2*) **unfolding** *ef* **using** *avF* **by** *simp*
note *mIH* = *mIH*[*OF this*]
thus *?thesis* **using** *avF* **by** *simp*
next
case *None*
obtain *a* **where** *Some*: *ifex-sat-list t* = *Some a* **using** *None IF.prems*(*1*) **by** *fastforce*
have *ef*: *u* = (*v*, *True*) # *a* **using** *IF.prems*(*1*) *Some None* **by** *simp*
have *avT*: *as v* = *True* **using** *IF.prems*(*2*) **unfolding** *ef* **by**(*simp split*: *if-splits*)
have *val-ifex* (*sat-list-to-bdt a*) *as* **using** *IF.prems*(*2*) **unfolding** *ef* **using** *avT* **by** *simp*
note *mIH* = *IF.IH*(*1*)[*OF Some this*]
thus *?thesis* **using** *avT* **by** *simp*
qed
qed *simp-all*

lemma *bf-ifex-rel-consts*[*simp*,*intro*!]:
(*bf-True*, *Trueif*) ∈ *bf-ifex-rel*
(*bf-False*, *Falseif*) ∈ *bf-ifex-rel*
by(*fastforce simp add*: *bf-ifex-rel-def*)+
lemma *bf-ifex-rel-lit*[*simp*,*intro*!]:
(*bf-lit v*, *IFC v Trueif Falseif*) ∈ *bf-ifex-rel*
by(*simp add*: *bf-ifex-rel-def IFC-def bf-lit-def*)

lemma *bf-ifex-rel-consts-ensured*[*simp*]:
(*bf-True*,*x*) ∈ *bf-ifex-rel* ⟷ (*x* = *Trueif*)
(*bf-False*,*x*) ∈ *bf-ifex-rel* ⟷ (*x* = *Falseif*)
by(*auto simp add*: *bf-ifex-rel-def*
intro: *roifex-Trueif-unique roifex-Falseif-unique*)

lemma *bf-ifex-rel-consts-ensured-rev*[*simp*]:
(*x*,*Trueif*) ∈ *bf-ifex-rel* ⟷ (*x* = *bf-True*)
(*x*,*Falseif*) ∈ *bf-ifex-rel* ⟷ (*x* = *bf-False*)
by(*simp-all add*: *bf-ifex-rel-def fun-eq-iff*)

declare *ifex-ite-opt.simps restrict-top.simps lowest-tops.simps*[*simp del*]

end

4 Option Helpers

These definitions were contributed by Peter Lammich.

theory *Option-Helpers*
imports *Main HOL−Library.Monad-Syntax*
begin

primrec *oassert* :: *bool* ⇒ *unit option* **where**
oassert True = *Some* () | *oassert False* = *None*

lemma *oassert-iff* [*simp*]:
oassert Φ = *Some x* ⟷ Φ
oassert Φ = *None* ⟷ ¬Φ
by (*cases* Φ) *auto*

The idea is that we want the result of some computation to be *Some v* and the contents of v to satisfy some property Q.

primrec *ospec* :: (′*a option*) ⇒ (′*a* ⇒ *bool*) ⇒ *bool* **where**
ospec None - = *False*
| *ospec* (*Some v*) *Q* = *Q v*

named-theorems *ospec-rules*

lemma *oreturn-rule*[*ospec-rules*]: ⟦ *P r* ⟧ ⟹ *ospec* (*Some r*) *P* **by** *simp*

lemma *obind-rule*[*ospec-rules*]: ⟦ *ospec m Q*; ⋀*r*. *Q r* ⟹ *ospec* (*f r*) *P* ⟧ ⟹ *ospec* (*m* ≫= *f*) *P*
apply (*cases m*)
apply (*auto split*: *Option.bind-splits*)
done

lemma *ospec-alt*: *ospec m P* = (*case m of None* ⇒ *False* | *Some x* ⇒ *P x*)
by (*auto split*: *option.splits*)

lemma *ospec-bind-simp*: *ospec* (*m* ≫= *f*) *P* ⟷ (*ospec m* (λ*r*. *ospec* (*f r*) *P*))
apply (*cases m*)
apply (*auto split*: *Option.bind-splits*)
done

lemma *ospec-cons*:
assumes *ospec m Q*
assumes ⋀*r*. *Q r* ⟹ *P r*
shows *ospec m P*
using *assms* **by** (*cases m*) *auto*

lemma *oreturn-synth*: *ospec* (*Some x*) (λ*r*. *r*=*x*) **by** *simp*

lemma *ospecD*: *ospec x P* ⟹ *x* = *Some y* ⟹ *P y* **by** *simp*
lemma *ospecD2*: *ospec x P* ⟹ ∃ *y*. *x* = *Some y* ∧ *P y* **by**(*cases x*) *simp-all*

end

5 Abstract ITE Implementation

theory *Abstract-Impl*
imports *BDT*

Automatic-Refinement.Refine-Lib
Option-Helpers
begin

datatype $('a, 'ni)$ *IFEXD* = *TD* | *FD* | *IFD* $'a$ $'ni$ $'ni$

locale *bdd-impl-pre* =
fixes $R :: 's \Rightarrow ('ni \times ('a :: linorder)\ ifex)\ set$
fixes $I :: 's \Rightarrow bool$
begin
definition *les*:: $'s \Rightarrow 's \Rightarrow bool$ **where**
$les\ s\ s' == \forall\, ni\ n.\ (ni, n) \in R\ s \longrightarrow (ni, n) \in R\ s'$
end

locale *bdd-impl* = *bdd-impl-pre R* **for** $R :: 's \Rightarrow ('ni \times ('a :: linorder)\ ifex)\ set$ +
fixes $Timpl :: 's \rightharpoonup ('ni \times 's)$
fixes $Fimpl :: 's \rightharpoonup ('ni \times 's)$
fixes $IFimpl :: 'a \Rightarrow 'ni \Rightarrow 'ni \Rightarrow 's \rightharpoonup ('ni \times 's)$
fixes $DESTRimpl :: 'ni \Rightarrow 's \rightharpoonup ('a, 'ni)\ IFEXD$

assumes *Timpl-rule*: $I\ s \Longrightarrow ospec\ (Timpl\ s)\ (\lambda(ni, s').\ (ni, Trueif) \in R\ s' \wedge I\ s' \wedge les\ s\ s')$
assumes *Fimpl-rule*: $I\ s \Longrightarrow ospec\ (Fimpl\ s)\ (\lambda(ni, s').\ (ni, Falseif) \in R\ s' \wedge I\ s' \wedge les\ s\ s')$
assumes *IFimpl-rule*: $[\![I\ s;\ (ni1,n1) \in R\ s;(ni2,n2) \in R\ s]\!]$
$\Longrightarrow ospec\ (IFimpl\ v\ ni1\ ni2\ s)\ (\lambda(ni, s').\ (ni, IFC\ v\ n1\ n2) \in R\ s' \wedge I\ s' \wedge les\ s\ s')$

assumes *DESTRimpl-rule1*: $I\ s \Longrightarrow (ni, Trueif) \in R\ s \Longrightarrow ospec\ (DESTRimpl\ ni\ s)\ (\lambda r.\ r = TD)$
assumes *DESTRimpl-rule2*: $I\ s \Longrightarrow (ni, Falseif) \in R\ s \Longrightarrow ospec\ (DESTRimpl\ ni\ s)\ (\lambda r.\ r = FD)$
assumes *DESTRimpl-rule3*: $I\ s \Longrightarrow (ni, IF\ v\ n1\ n2) \in R\ s \Longrightarrow$
$ospec\ (DESTRimpl\ ni\ s)$
$(\lambda r.\ \exists\, ni1\ ni2.\ r = (IFD\ v\ ni1\ ni2) \wedge (ni1, n1) \in R\ s \wedge (ni2, n2) \in R\ s)$
begin

lemma *les-refl*[*simp*,*intro*!]:$les\ s\ s$ **by** (*auto simp add*: *les-def*)
lemma *les-trans*[*trans*]:$les\ s1\ s2 \Longrightarrow les\ s2\ s3 \Longrightarrow les\ s1\ s3$ **by** (*auto simp add*: *les-def*)
lemmas *DESTRimpl-rules* = *DESTRimpl-rule1 DESTRimpl-rule2 DESTRimpl-rule3*

lemma *DESTRimpl-rule-useless*:
$I\ s \Longrightarrow (ni, n) \in R\ s \Longrightarrow ospec\ (DESTRimpl\ ni\ s)\ (\lambda r.\ (case\ r\ of$
$TD \Rightarrow (ni, Trueif) \in R\ s\ |$
$FD \Rightarrow (ni, Falseif) \in R\ s\ |$
$IFD\ v\ nt\ ne \Rightarrow (\exists\, t\ e.\ n = IF\ v\ t\ e \wedge (ni, IF\ v\ t\ e) \in R\ s)))$
by(*cases n*; *clarify*; *drule* (*1*) *DESTRimpl-rules*; *drule ospecD2*; *clarsimp*)

lemma *DESTRimpl-rule*:
  I s $\Longrightarrow$ (*ni*, *n*) $\in$ *R s* $\Longrightarrow$ *ospec* (*DESTRimpl ni s*) (λ*r*. (*case n of*
    Trueif $\Rightarrow$ *r* = *TD* |
    Falseif $\Rightarrow$ *r* = *FD* |
    IF v t e $\Rightarrow$ ($\exists$ *tn en*. *r* = *IFD v tn en* $\wedge$ (*tn*,*t*) $\in$ *R s* $\wedge$ (*en*,*e*) $\in$ *R s*)))
by(*cases n*; *clarify*; *drule* (*1*) *DESTRimpl-rules*; *drule ospecD2*; *clarsimp*)

definition *case-ifexi fti ffi fii ni s* $\equiv$ *do* {
  dest $\leftarrow$ *DESTRimpl ni s*;
  case dest of
    TD $\Rightarrow$ *fti s*
  | *FD* $\Rightarrow$ *ffi s*
  | *IFD v ti ei* $\Rightarrow$ *fii v ti ei s*}

lemma *case-ifexi-rule*:
  assumes *INV*: *I s*
  assumes *NI*: (*ni*,*n*)$\in$*R s*
  assumes *FTI*: $[\![$ *n* = *Trueif* $]\!]$ $\Longrightarrow$ *ospec* (*fti s*) (λ(*r*,*s'*). (*r*,*ft*) $\in$ *Q s* $\wedge$ *I' s'*)
  assumes *FFI*: $[\![$ *n* = *Falseif* $]\!]$ $\Longrightarrow$ *ospec* (*ffi s*) (λ(*r*,*s'*). (*r*,*ff*) $\in$ *Q s* $\wedge$ *I' s'*)
  assumes *FII*: $\bigwedge$*ti ei v t e*. $[\![$ *n* = *IF v t e*; (*ti*,*t*)$\in$*R s*; (*ei*,*e*)$\in$*R s* $]\!]$ $\Longrightarrow$ *ospec* (*fii v ti ei s*) (λ(*r*,*s'*). (*r*,*fi v t e*) $\in$ *Q s* $\wedge$ *I' s'*)
  shows *ospec* (*case-ifexi fti ffi fii ni s*) (λ(*r*,*s'*). (*r*,*case-ifex ft ff fi n*) $\in$ *Q s* $\wedge$ *I' s'*)
  unfolding *case-ifexi-def*
  apply (*cases n*)
    subgoal
      apply (*rule obind-rule*)
       apply (*rule DESTRimpl-rule1*[*OF INV*])
       using *NI FTI* **by** (*auto*)
    subgoal
      apply (*rule obind-rule*)
       apply (*rule DESTRimpl-rule2*[*OF INV*])
       using *NI FFI* **by** (*auto*)
    subgoal
      apply (*rule obind-rule*)
       apply (*rule DESTRimpl-rule3*[*OF INV*])
       using *NI FII* **by** (*auto*)
done

abbreviation *return x* $\equiv$ λ*s*. *Some* (*x*,*s*)

primrec *lowest-tops-impl* **where**
lowest-tops-impl [] *s* = *Some* (*None*,*s*) |
lowest-tops-impl (*e*#*es*) *s* =
    case-ifexi
      (λ*s*. *lowest-tops-impl es s*)
      (λ*s*. *lowest-tops-impl es s*)
      (λ*v t e s*. *do* {
      (*rec*,*s*) $\leftarrow$ *lowest-tops-impl es s*;

```
    (case rec of
      Some u ⇒ Some ((Some (min u v)), s) |
      None ⇒ Some ((Some v), s))
    }) e s

declare lowest-tops-impl.simps[simp del]

fun lowest-tops-alt where
lowest-tops-alt [] = None |
lowest-tops-alt (e#es) = (
  let rec = lowest-tops-alt es in
  case-ifex
    rec
    rec
    (λv t e. (case rec of
        Some u ⇒ (Some (min u v)) |
        None ⇒ (Some v))
    ) e
  )

lemma lowest-tops-alt: lowest-tops l = lowest-tops-alt l
  by (induction l rule: lowest-tops.induct) (auto split: option.splits simp: lowest-tops.simps)

lemma lowest-tops-impl-R:
  assumes list-all2 (in-rel (R s)) li l I s
  shows ospec (lowest-tops-impl li s) (λ(r,s'). r = lowest-tops l ∧ s'=s)
  unfolding lowest-tops-alt
  using assms apply (induction rule: list-all2-induct)
   subgoal by (simp add: lowest-tops-impl.simps)
   subgoal
     apply (simp add: lowest-tops-impl.simps)
     apply (rule case-ifexi-rule[where Q=λs. Id, unfolded pair-in-Id-conv])
      apply assumption+
     apply (rule obind-rule)
      apply assumption
     apply (clarsimp split: option.splits)
    done
  done


definition restrict-top-impl where
restrict-top-impl e vr vl s =
  case-ifexi
    (return e)
    (return e)
    (λv te ee. return (if v = vr then (if vl then te else ee) else e))
    e s
```

```
lemma restrict-top-alt: restrict-top n var val = (case n of
  (IF v t e) ⇒ (if v = var then (if val then t else e) else (IF v t e))
| - ⇒ n)
  apply (induction n var val rule: restrict-top.induct)
  apply (simp-all)
  done

lemma restrict-top-impl-spec: I s ⟹ (ni,n) ∈ R s ⟹ ospec (restrict-top-impl ni
vr vl s) (λ(res,s′). (res, restrict-top n vr vl) ∈ R s ∧ s′=s)
  unfolding restrict-top-impl-def restrict-top-alt
  by (rule case-ifexi-rule[where I′=λs′. s′=s and Q=R, simplified]) auto


partial-function(option) ite-impl where
ite-impl i t e s = do {
  (lt,-) ← lowest-tops-impl [i, t, e] s;
  (case lt of
    Some a ⇒ do {
      (ti,-) ← restrict-top-impl i a True s;
      (tt,-) ← restrict-top-impl t a True s;
      (te,-) ← restrict-top-impl e a True s;
      (fi,-) ← restrict-top-impl i a False s;
      (ft,-) ← restrict-top-impl t a False s;
      (fe,-) ← restrict-top-impl e a False s;
      (tb,s) ← ite-impl ti tt te s;
      (fb,s) ← ite-impl fi ft fe s;
      IFimpl a tb fb s}
  | None ⇒ case-ifexi (λ-.(Some (t,s))) (λ-.(Some (e,s))) (λ- - - -. None) i s
)}

lemma ite-impl-R: I s
      ⟹ in-rel (R s) ii i ⟹ in-rel (R s) ti t ⟹ in-rel (R s) ei e
      ⟹ ospec (ite-impl ii ti ei s) (λ(r, s′). (r, ifex-ite i t e) ∈ R s′ ∧ I s′ ∧ les s
s′)
proof(induction i t e arbitrary: s ii ti ei rule: ifex-ite.induct, goal-cases)
  case (1 i t e s ii ti ei) note goal1 = 1
  have la2: list-all2 (in-rel (R s)) [ii,ti,ei] [i,t,e] using goal1(4−6) by simp
  show ?case proof(cases lowest-tops [i,t,e])
    case None from goal1(3−6) show ?thesis
        apply(subst ite-impl.simps)
        apply(rule obind-rule[where Q=λ(r, s′). r = lowest-tops [i,t,e] ∧ s′=s])
         apply(rule ospec-cons)
          apply(rule lowest-tops-impl-R[OF la2])
          apply(assumption)
         apply(clarsimp split: prod.splits)
        apply(simp add: None split: prod.splits)
        apply(clarsimp)
        apply(rule ospec-cons)
```

```
        apply(rule case-ifexi-rule[where I′=λs′. s′=s])
       using None by (auto split: prod.splits ifex.splits simp: lowest-tops.simps)
    next
    case (Some lv)
      note mIH = goal1(1,2)[OF Some]
      from goal1(3−6) show ?thesis
        apply(subst ite-impl.simps)
        apply(rule obind-rule[where Q=λ(r, s′). r = lowest-tops [i,t,e]])
         apply(rule ospec-cons)
          apply(rule lowest-tops-impl-R[OF la2])
          apply(assumption)
         apply(clarsimp split: prod.splits)
        apply(simp add: Some split: prod.splits)
        apply(clarsimp)

         apply(rule obind-rule, rule restrict-top-impl-spec, assumption+, clarsimp
split: prod.splits)+
        apply(rule obind-rule)
         apply(rule mIH(1))
           apply(simp;fail)+
        apply(clarsimp)
        apply(rule obind-rule)
         apply(rule mIH(2))
           apply(simp add: les-def;fail)+
        apply(simp split: prod.splits)
        apply(rule ospec-cons)
         apply(rule IFimpl-rule)
          apply(simp add: les-def;fail)+
          using les-def les-trans by blast+
    qed
qed

lemma case-ifexi-mono[partial-function-mono]:
  assumes [partial-function-mono]:
    mono-option (λF. fti F s)
    mono-option (λF. ffi F s)
    ⋀x31 x32 x33. mono-option (λF. fii F x31 x32 x33 s)
  shows mono-option (λF. case-ifexi (fti F) (ffi F) (fii F) ni s)
  unfolding case-ifexi-def by (tactic ‹Partial-Function.mono-tac @{context} 1›)

partial-function(option) val-impl :: ′ni ⇒ (′a ⇒ bool) ⇒ ′s ⇒ (bool×′s) option
where
val-impl e ass s = case-ifexi
  (λs. Some (True,s))
  (λs. Some (False,s))
  (λv t e s. val-impl (if ass v then t else e) ass s)
  e s

lemma I s ⟹ (ni,n) ∈ R s ⟹ ospec (val-impl ni ass s) (λ(r,s′). r = (val-ifex n
```

ass) ∧ *s′*=*s*)
  apply (*induction n arbitrary*: *ni*)
  subgoal
   apply (*subst val-impl.simps*)
   apply (*rule ospec-cons*)
    apply (*rule case-ifexi-rule*[**where** *I′*=λ*s′*. *s′*=*s* **and** *Q*=λ*s*. *Id*]; *assumption?*)
     by *auto*
  subgoal
   apply (*subst val-impl.simps*)
   apply (*rule ospec-cons*)
    apply (*rule case-ifexi-rule*[**where** *I′*=λ*s′*. *s′*=*s* **and** *Q*=λ*s*. *Id*]; *assumption?*)
     by *auto*
  subgoal
   apply (*subst val-impl.simps*)
   apply (*subst val-ifex.simps*)
   apply (*clarsimp*; *intro impI conjI*)
    apply (*rule ospec-cons*)
     apply (*rule case-ifexi-rule*[**where** *I′*=λ*s′*. *s′*=*s* **and** *Q*=λ*s*. *Id*]; *assumption?*)
       apply (*simp*; *fail*)
      apply (*simp*; *fail*)
     apply (*rule ospec-cons*)
      apply (*rprems*; *simp*; *fail*)
     apply (*simp*; *fail*)
    apply (*simp*; *fail*)
   apply (*rule ospec-cons*)
    apply (*rule case-ifexi-rule*[**where** *I′*=λ*s′*. *s′*=*s* **and** *Q*=λ*s*. *Id*]; *assumption?*)
      apply (*simp*; *fail*)
     apply (*simp*; *fail*)
    apply(*simp*)
    apply (*rule ospec-cons*)
     apply (*rprems*; *simp*; *fail*)
    apply (*simp*; *fail*)
   apply (*simp*; *fail*)
   done
  done

end

locale *bdd-impl-cmp-pre* = *bdd-impl-pre*
begin

definition *map-invar-impl m s* =
  (∀ *ii ti ei ri*. *m* (*ii*,*ti*,*ei*) = *Some ri* ⟶
  (∃ *i t e*. ((*ri*,*ifex-ite-opt i t e*) ∈ *R s*) ∧ (*ii*,*i*) ∈ *R s* ∧ (*ti*,*t*) ∈ *R s* ∧ (*ei*,*e*) ∈ *R s*))

lemma *map-invar-impl-les*: *map-invar-impl m s* ⟹ *les s s′* ⟹ *map-invar-impl m s′*
  unfolding *map-invar-impl-def bdd-impl-pre.les-def* **by** *blast*

lemma *map-invar-impl-update*: *map-invar-impl m s* $\Longrightarrow$
$(ii,i) \in R\ s \Longrightarrow (ti,t) \in R\ s \Longrightarrow (ei,e) \in R\ s \Longrightarrow$
$(ri,\ \textit{ifex-ite-opt}\ i\ t\ e) \in R\ s \Longrightarrow \textit{map-invar-impl}\ (m((ii,ti,ei) \mapsto ri))\ s$
unfolding *map-invar-impl-def* **by** *auto*

end

locale *bdd-impl-cmp* = *bdd-impl* + *bdd-impl-cmp-pre* +
fixes $M :: 'a \Rightarrow ('b \times 'b \times 'b) \Rightarrow 'b\ option$
fixes $U :: 'a \Rightarrow ('b \times 'b \times 'b) \Rightarrow 'b \Rightarrow 'a$
fixes $cmp :: 'b \Rightarrow 'b \Rightarrow bool$
assumes *cmp-rule1*: $I\ s \Longrightarrow (ni,\ i) \in R\ s \Longrightarrow (ni',\ i) \in R\ s \Longrightarrow cmp\ ni\ ni'$
assumes *cmp-rule2*: $I\ s \Longrightarrow cmp\ ni\ ni' \Longrightarrow (ni,\ i) \in R\ s \Longrightarrow (ni',\ i') \in R\ s \Longrightarrow i = i'$
assumes *map-invar-rule1*: $I\ s \Longrightarrow \textit{map-invar-impl}\ (M\ s)\ s$
assumes *map-invar-rule2*: $I\ s \Longrightarrow (ii,it) \in R\ s \Longrightarrow (ti,tt) \in R\ s \Longrightarrow (ei,et) \in R\ s \Longrightarrow$
$(ri,\ \textit{ifex-ite-opt}\ it\ tt\ et) \in R\ s \Longrightarrow U\ s\ (ii,ti,ei)\ ri = s' \Longrightarrow I\ s'$
assumes *map-invar-rule3*: $I\ s \Longrightarrow R\ (U\ s\ (ii,\ ti,\ ei)\ ri) = R\ s$
begin

lemma *cmp-rule-eq*: $I\ s \Longrightarrow (ni,\ i) \in R\ s \Longrightarrow (ni',\ i') \in R\ s \Longrightarrow cmp\ ni\ ni' \longleftrightarrow i = i'$
using *cmp-rule1 cmp-rule2* **by** *force*

lemma *DESTRimpl-Some*: $I\ s \Longrightarrow (ni,\ i) \in R\ s \Longrightarrow ospec\ (DESTRimpl\ ni\ s)\ (\lambda r.\ True)$
apply(*cases i*)
apply(*auto intro*: *ospec-cons dest*: *DESTRimpl-rules*)
done

fun *param-opt-impl* **where**
param-opt-impl i t e s = *do* {
$ii \leftarrow DESTRimpl\ i\ s$;
$ti \leftarrow DESTRimpl\ t\ s$;
$ei \leftarrow DESTRimpl\ e\ s$;
$(tn,s) \leftarrow Timpl\ s$;
$(fn,s) \leftarrow Fimpl\ s$;
Some ((*if* $ii = TD$ *then Some t else*
if $ii = FD$ *then Some e else*
if $ti = TD \land ei = FD$ *then Some i else*
if cmp t e then Some t else
if $ei = TD \land cmp\ i\ t$ *then Some tn else*
if $ti = FD \land cmp\ i\ e$ *then Some fn else*
None), *s*)}

declare *param-opt-impl.simps*[*simp del*]

lemma *param-opt-impl-lesI*:
  assumes *I s* (*ii*,*i*) ∈ *R s* (*ti*,*t*) ∈ *R s* (*ei*,*e*) ∈ *R s*
  shows *ospec* (*param-opt-impl ii ti ei s*)
          (λ(*r*,*s*′). *I s*′ ∧ *les s s*′)
  using *assms* **apply**(*subst param-opt-impl.simps*)
  by (*auto simp add*: *param-opt-def les-def intro*!: *obind-rule*
          dest: *DESTRimpl-Some Timpl-rule Fimpl-rule*)

lemma *param-opt-impl-R*:
  assumes *I s* (*ii*,*i*) ∈ *R s* (*ti*,*t*) ∈ *R s* (*ei*,*e*) ∈ *R s*
  shows *ospec* (*param-opt-impl ii ti ei s*)
          (λ(*r*,*s*′). *case r of None* ⇒ *param-opt i t e* = *None*
                    | *Some r* ⇒ (∃ *r*′. *param-opt i t e* = *Some r*′ ∧ (*r*, *r*′)
∈ *R s*′))
  using *assms* **apply**(*subst param-opt-impl.simps*)
  apply(*rule obind-rule*)
   apply(*rule DESTRimpl-rule*; *assumption*)
  apply(*rule obind-rule*)
   apply(*rule DESTRimpl-rule*; *assumption*)
  apply(*rule obind-rule*)
   apply(*rule DESTRimpl-rule*; *assumption*)
  apply(*rule obind-rule*)
   apply(*rule Timpl-rule*; *assumption*)
  apply(*safe*)
  apply(*rule obind-rule*)
   apply(*rule Fimpl-rule*; *assumption*)
  by (*auto simp add*: *param-opt-def les-def cmp-rule-eq split*: *ifex.splits*)

partial-function(*option*) *ite-impl-opt* **where**
ite-impl-opt i t e s = *do* {
  (*ld*, *s*) ← *param-opt-impl i t e s*;
  (*case ld of Some b* ⇒ *Some* (*b*, *s*) |
  None ⇒
  do {
  (*lt*,-) ← *lowest-tops-impl* [*i*, *t*, *e*] *s*;
  (*case lt of*
    Some a ⇒ *do* {
      (*ti*,-) ← *restrict-top-impl i a True s*;
      (*tt*,-) ← *restrict-top-impl t a True s*;
      (*te*,-) ← *restrict-top-impl e a True s*;
      (*fi*,-) ← *restrict-top-impl i a False s*;
      (*ft*,-) ← *restrict-top-impl t a False s*;
      (*fe*,-) ← *restrict-top-impl e a False s*;
      (*tb*,*s*) ← *ite-impl-opt ti tt te s*;
      (*fb*,*s*) ← *ite-impl-opt fi ft fe s*;
      IFimpl a tb fb s}
  | *None* ⇒ *case-ifexi* (λ-.(*Some* (*t*,*s*))) (λ-.(*Some* (*e*,*s*))) (λ- - - -. *None*) *i s*
)})}

lemma *ospec-and*: *ospec f P* ⟹ *ospec f Q* ⟹ *ospec f* (λ*x*. *P x* ∧ *Q x*)
  using *ospecD2* **by** *force*

lemma *ite-impl-opt-R*:
  I s
  ⟹ *in-rel* (*R s*) *ii i* ⟹ *in-rel* (*R s*) *ti t* ⟹ *in-rel* (*R s*) *ei e*
  ⟹ *ospec* (*ite-impl-opt ii ti ei s*) (λ(*r*, *s'*). (*r*, *ifex-ite-opt i t e*) ∈ *R s'* ∧ *I s'* ∧ *les s s'*)
proof(*induction i t e arbitrary*: *s ii ti ei rule*: *ifex-ite-opt.induct*, *goal-cases*)
  note *ifex-ite-opt.simps*[*simp del*] *restrict-top.simps*[*simp del*]
  case (*1 i t e s ii ti ei*) **note** *goal1* = *1*
  have *la2*: *list-all2* (*in-rel* (*R s*)) [*ii*,*ti*,*ei*] [*i*,*t*,*e*] **using** *goal1*(*4*−*6*) **by** *simp*
  note *mIH* = *goal1*(*1*,*2*)
  from *goal1*(*3*−*6*) **show** *?case*
    apply(*cases param-opt i t e*)
     defer
     apply(*subst ite-impl-opt.simps*)
      apply(*rule obind-rule*)
       apply(*rule ospec-and*[*OF param-opt-impl-R param-opt-impl-lesI*])
             apply(*auto simp add*: *les-def ifex-ite-opt.simps split*: *option.splits*)[*9*]

    apply(*frule param-opt-lowest-tops-lem*)
    apply(*clarsimp*)
    apply(*subst ite-impl-opt.simps*)
     apply(*rule obind-rule*)
     apply(*rule ospec-and*[*OF param-opt-impl-R param-opt-impl-lesI*])
           apply(*auto split*: *option.splits*)[*8*]
    apply(*clarsimp split*: *option.splits*)
    apply(*rule obind-rule*[**where** *Q*=λ(*r*, *s'*). *r* = *lowest-tops* [*i*,*t*,*e*]])
     apply(*rule ospec-cons*)
      apply(*rule lowest-tops-impl-R*)
       using *les-def* **apply**(*fastforce*)
      apply(*assumption*)
     apply(*fastforce*)
    using *BDT.param-opt-lowest-tops-lem* **apply**(*clarsimp split*: *prod.splits*)

    apply(*rule obind-rule*, *rule restrict-top-impl-spec*, *assumption*, *auto simp add*: *les-def split*: *prod.splits*)+
    apply(*rule obind-rule*)
     apply(*rule mIH*(*1*))
         apply(*simp add*: *les-def*;*fail*)+
    apply(*clarsimp*)
     apply(*rule obind-rule*)
     apply(*rule mIH*(*2*))
         apply(*simp add*: *les-def*;*fail*)+
    apply(*simp add*: *ifex-ite-opt.simps split*: *prod.splits*)
    apply(*rule ospec-cons*)
     apply(*rule IFimpl-rule*)

apply(*auto simp add*: *les-def*;*fail*)+
done
qed

partial-function(*option*) *ite-impl-lu* **where**
ite-impl-lu i t e s = *do* {
(*case M s* (*i*,*t*,*e*) *of Some b* ⇒ *Some* (*b*,*s*) | *None* ⇒ *do* {
(*ld*, *s*) ← *param-opt-impl i t e s*;
(*case ld of Some b* ⇒ *Some* (*b*, *s*) |
None ⇒
do {
(*lt*,-) ← *lowest-tops-impl* [*i*, *t*, *e*] *s*;
(*case lt of*
Some a ⇒ *do* {
(*ti*,-) ← *restrict-top-impl i a True s*;
(*tt*,-) ← *restrict-top-impl t a True s*;
(*te*,-) ← *restrict-top-impl e a True s*;
(*fi*,-) ← *restrict-top-impl i a False s*;
(*ft*,-) ← *restrict-top-impl t a False s*;
(*fe*,-) ← *restrict-top-impl e a False s*;
(*tb*,*s*) ← *ite-impl-lu ti tt te s*;
(*fb*,*s*) ← *ite-impl-lu fi ft fe s*;
(*r*,*s*) ← *IFimpl a tb fb s*;
let s = *U s* (*i*,*t*,*e*) *r*;
Some (*r*,*s*)
} |
None ⇒ *None*
)})})}

declare *ifex-ite-opt.simps*[*simp del*]

lemma *ite-impl-lu-R*: *I s*
$\Longrightarrow$ (*ii*,*i*) ∈ *R s* $\Longrightarrow$ (*ti*,*t*) ∈ *R s* $\Longrightarrow$ (*ei*,*e*) ∈ *R s*
$\Longrightarrow$ *ospec* (*ite-impl-lu ii ti ei s*)
(λ(*r*, *s'*). (*r*, *ifex-ite-opt i t e*) ∈ *R s'* ∧ *I s'* ∧ *les s s'*)
proof(*induction i t e arbitrary*: *s ii ti ei rule*: *ifex-ite-opt.induct*, *goal-cases*)
note *restrict-top.simps*[*simp del*]
case (*1 i t e s ii ti ei*) **note** *goal1* = *1*
have *la2*: *list-all2* (*in-rel* (*R s*)) [*ii*,*ti*,*ei*] [*i*,*t*,*e*] **using** *goal1*(*4*−*6*) **by** *simp*
note *mIH* = *goal1*(*1*,*2*)
from *goal1*(*3*−*6*) **show** *?case*
apply(*subst ite-impl-lu.simps*)
apply(*cases M s* (*ii*, *ti*, *ei*))
defer

apply(*frule map-invar-rule1*)
apply(*simp only*: *option.simps ospec.simps prod.simps simp-thms les-refl*)
apply(*subst* (*asm*) *map-invar-impl-def*)
apply(*erule allE*[**where** *x* = *ii*])

apply(*erule allE*[**where** *x* = *ti*])
apply(*erule allE*[**where** *x* = *ei*])
apply(*rename-tac a*)
apply(*erule-tac x* = *a* **in** *allE*)
apply(*metis cmp-rule-eq*)

apply(*clarsimp*)
apply(*cases param-opt i t e*)
defer

apply(*rule obind-rule*)
apply(*rule ospec-and*[*OF param-opt-impl-R param-opt-impl-lesI*])
apply(*auto simp add*: *map-invar-impl-les ifex-ite-opt.simps split*: *option.splits*)[*9*]

apply(*frule param-opt-lowest-tops-lem*)
apply(*clarsimp*)
apply(*rule obind-rule*)
apply(*rule ospec-and*[*OF param-opt-impl-R param-opt-impl-lesI*])
apply(*auto split*: *option.splits*)[*8*]
apply(*clarsimp split*: *option.splits*)
apply(*rule-tac obind-rule*[**where** *Q*=λ(*r*, *s'*). *r* = *lowest-tops* [*i*,*t*,*e*]])
apply(*rule ospec-cons*)
apply(*rule lowest-tops-impl-R*)
using *les-def* **apply**(*fastforce*)
apply(*assumption*)
apply(*fastforce*)
using *BDT.param-opt-lowest-tops-lem* **apply**(*clarsimp split*: *prod.splits*)
apply(*rule obind-rule*, *rule restrict-top-impl-spec*, *assumption*+, *auto simp add*: *les-def split*: *prod.splits*)+
apply(*rule obind-rule*)
apply(*rule mIH*(*1*))
apply(*simp add*: *map-invar-impl-les les-def*;*fail*)+
apply(*clarsimp*)
apply(*rule obind-rule*)
apply(*rule mIH*(*2*))
apply(*simp add*: *map-invar-impl-les les-def*;*fail*)+
apply(*simp add*: *ifex-ite-opt.simps split*: *prod.splits*)
apply(*rule obind-rule*)
apply(*rule IFimpl-rule*)
apply(*simp*)
apply(*auto simp add*: *les-def*)[*2*]
apply(*clarsimp simp add*: *les-def*)
apply(*safe*)
using *map-invar-rule3* **apply**(*presburger*)
apply(*rule map-invar-rule2*)
prefer *6* **apply**(*blast*)
apply(*blast*)
apply(*blast*)

apply(*blast*)
apply(*blast*)
apply(*clarsimp simp add*: *ifex-ite-opt.simps*)
using *map-invar-rule3* **by** *presburger*
qed

end
end

6 Pointermap

theory *Pointer-Map*
imports *Main*
begin

We need a datastructure that supports the following two operations:

- Given an element, it can construct a pointer (i.e., a small representation) of that element. It will always construct the same pointer for equal elements.
- Given a pointer, we can retrieve the element

record *'a pointermap* =
entries :: *'a list*
getentry :: *'a* ⇒ *nat option*

definition *pointermap-sane m* ≡ (*distinct* (*entries m*) ∧
(∀ *n* ∈ {..<*length* (*entries m*)}. *getentry m* (*entries m* ! *n*) = *Some n*) ∧
(∀ *p i*. *getentry m p* = *Some i* ⟶ *entries m* ! *i* = *p* ∧ *i* < *length* (*entries m*)))

definition *empty-pointermap* ≡ ⦇*entries* = [], *getentry* = λ*p*. *None* ⦈
lemma *pointermap-empty-sane*[*simp*, *intro*!]: *pointermap-sane empty-pointermap*
unfolding *empty-pointermap-def pointermap-sane-def* **by** *simp*

definition *pointermap-insert a m* ≡ ⦇*entries* = (*entries m*)@[*a*], *getentry* = (*getentry m*)(*a* ↦ *length* (*entries m*))⦈

definition *pm-pth m p* ≡ *entries m* ! *p*

definition *pointermap-p-valid p m* ≡ *p* < *length* (*entries m*)

definition *pointermap-getmk a m* ≡ (*case getentry m a of Some p* ⇒ (*p*,*m*) | *None* ⇒ *let u* = *pointermap-insert a m in* (*the* (*getentry u a*), *u*))

lemma *pointermap-sane-appendD*: *pointermap-sane s* ⟹ *m* ∉ *set* (*entries s*) ⟹ *pointermap-sane* (*pointermap-insert m s*)
unfolding *pointermap-sane-def pointermap-insert-def*
proof(*intro conjI*[*rotated*],*goal-cases*)

case *3* **thus** *?case* **by** *simp*
next
case *2*
{
fix *n*
have ⟦*distinct* (*entries s*) ∧ (∀ *x*. *x* ∈ {..<*length* (*entries s*)} ⟶ *getentry s* (*entries s* ! *x*) = *Some x*) ∧ (∀ *p i*. *getentry s p* = *Some i* ⟶ *entries s* ! *i* = *p* ∧ *i* < *length* (*entries s*)); *m* ∉ *set* (*entries s*);
n ∈ {..<*length* (*entries* (|*entries* = *entries s* @ [*m*], *getentry* = (*getentry s*)(*m* ↦ *length* (*entries s*))|))}; *n* < *length* (*entries s*)⟧
⟹ *getentry* (|*entries* = *entries s* @ [*m*], *getentry* = (*getentry s*)(*m* ↦ *length* (*entries s*))|) (*entries* (|*entries* = *entries s* @ [*m*], *getentry* = (*getentry s*)(*m* ↦ *length* (*entries s*))|) ! *n*) = *Some n*
⟦*distinct* (*entries s*) ∧ (∀ *x*. *x* ∈ {..<*length* (*entries s*)} ⟶ (*getentry s*) (*entries s* ! *x*) = *Some x*) ∧ (∀ *p i*. *getentry s p* = *Some i* ⟶ *entries s* ! *i* = *p* ∧ *i* < *length* (*entries s*)); *m* ∉ *set* (*entries s*);
n ∈ {..<*length* (*entries* (|*entries* = *entries s* @ [*m*], *getentry* = (*getentry s*)(*m* ↦ *length* (*entries s*))|))}; ¬ *n* < *length* (*entries s*)⟧
⟹ *getentry* (|*entries* = *entries s* @ [*m*], *getentry* = (*getentry s*)(*m* ↦ *length* (*entries s*))|) (*entries* (|*entries* = *entries s* @ [*m*], *getentry* = (*getentry s*)(*m* ↦ *length* (*entries s*))|) ! *n*) = *Some n*
proof(*goal-cases*)
case *1* **note** *goal1* = *1*
from *goal1*(*4*) **have** *sa*: ⋀*a*. (*entries s* @ *a*) ! *n* = *entries s* ! *n* **by** (*simp add*: *nth-append*)
from *goal1*(*1*,*4*) **have** *ih*: *getentry s* (*entries s* ! *n*) = *Some n* **by** *simp*
from *goal1*(*2*,*4*) **have** *ne*: *entries s* ! *n* ≠ *m* **using** *nth-mem* **by** *fastforce*
from *sa ih ne* **show** *?case* **by** *simp*
next
case *2* **note** *goal2* = *2*
from *goal2*(*3*,*4*) **have** *ln*: *n* = *length* (*entries s*) **by** *simp*
hence *sa*: ⋀*a*. (*entries s* @ [*a*]) ! *n* = *a* **by** *simp*
from *sa ln* **show** *?case* **by** *simp*
qed
} **note** *h* = *this*
with *2* **show** *?case* **by** *blast*

next
case *1* **thus** *?case*
by(*clarsimp simp add*: *nth-append fun-upd-same Ball-def*) *force*
qed

lemma *luentries-noneD*: *getentry s a* = *None* ⟹ *pointermap-sane s* ⟹ *a* ∉ *set* (*entries s*)
unfolding *pointermap-sane-def*
proof(*rule ccontr*, *goal-cases*)
case *1*
from *1*(*3*) **obtain** *n* **where** *n* < *length* (*entries s*) *entries s* ! *n* = *a* **unfolding** *in-set-conv-nth* **by** *blast*

with *1*(*2*,*1*) **show** *False* **by** *force*
qed

lemma *pm-pth-append*: *pointermap-p-valid p m* ⟹ *pm-pth* (*pointermap-insert a m*) *p* = *pm-pth m p*
unfolding *pointermap-p-valid-def pm-pth-def pointermap-insert-def*
by(*simp add*: *nth-append*)

lemma *pointermap-insert-in*: *u* = (*pointermap-insert a m*) ⟹ *pm-pth u* (*the* (*getentry u a*)) = *a*
unfolding *pointermap-insert-def pm-pth-def*
by(*simp*)

lemma *pointermap-insert-p-validI*: *pointermap-p-valid p m* ⟹ *pointermap-p-valid p* (*pointermap-insert a m*)
unfolding *pointermap-insert-def pointermap-p-valid-def*
by *simp*

thm *nth-eq-iff-index-eq*
lemma *pth-eq-iff-index-eq*: *pointermap-sane m* ⟹ *pointermap-p-valid p1 m* ⟹ *pointermap-p-valid p2 m* ⟹ (*pm-pth m p1* = *pm-pth m p2*) ⟷ (*p1* = *p2*)
unfolding *pointermap-sane-def pointermap-p-valid-def pm-pth-def*
using *nth-eq-iff-index-eq* **by** *blast*

lemma *pointermap-p-valid-updateI*: *pointermap-sane m* ⟹ *getentry m a* = *None* ⟹ *u* = *pointermap-insert a m* ⟹ *p* = *the* (*getentry u a*) ⟹ *pointermap-p-valid p u*
by(*simp add*: *pointermap-sane-def pointermap-p-valid-def pointermap-insert-def*)

lemma *pointermap-get-validI*: *pointermap-sane m* ⟹ *getentry m a* = *Some p* ⟹ *pointermap-p-valid p m*
by(*simp add*: *pointermap-sane-def pointermap-p-valid-def*)

lemma *pointermap-sane-getmkD*:
assumes *sn*: *pointermap-sane m*
assumes *res*: *pointermap-getmk a m* = (*p*,*u*)
shows *pointermap-sane u* ∧ *pointermap-p-valid p u*
using *sn res*[*symmetric*]
apply(*cases getentry m a*)
apply(*simp-all add*: *pointermap-getmk-def Let-def split*: *option.split*)
apply(*rule*)
apply(*rule pointermap-sane-appendD*)
apply(*clarify*;*fail*)+
apply(*rule luentries-noneD*)
apply(*clarify*;*fail*)+
apply(*rule pointermap-p-valid-updateI*[*OF - - refl refl*])
apply(*clarify*;*fail*)+
apply(*erule pointermap-get-validI*)
by *simp*

lemma *pointermap-update-pthI*:
  assumes *sn*: *pointermap-sane m*
  assumes *res*: *pointermap-getmk a m* = (*p*,*u*)
  shows *pm-pth u p* = *a*
using *assms*
  apply(*simp add*: *pointermap-getmk-def Let-def split*: *option.splits*)
   apply(*meson pointermap-insert-in*)
  apply(*clarsimp simp*: *pointermap-sane-def pm-pth-def*)
done

lemma *pointermap-p-valid-inv*:
  assumes *pointermap-p-valid p m*
  assumes *pointermap-getmk a m* = (*x*,*u*)
  shows *pointermap-p-valid p u*
using *assms*
by(*simp add*: *pointermap-getmk-def Let-def split*: *option.splits*) (*meson pointermap-insert-p-validI*)

lemma *pointermap-p-pth-inv*:
  assumes *pv*: *pointermap-p-valid p m*
  assumes *u*: *pointermap-getmk a m* = (*x*,*u*)
  shows *pm-pth u p* = *pm-pth m p*
using *pm-pth-append*[*OF pv*] *u*
by(*clarsimp simp*: *pointermap-getmk-def Let-def split*: *option.splits*)

lemma *pointermap-backward-valid*:
  assumes *puv*: *pointermap-p-valid p u*
  assumes *u*: *pointermap-getmk a m* = (*x*,*u*)
  assumes *ne*: $x \neq p$
  shows *pointermap-p-valid p m*

using *assms*
by (*auto simp*: *Let-def pointermap-getmk-def pointermap-p-valid-def pointermap-insert-def split*: *option.splits*)

end

7 Functional interpretation for the abstract implementation

theory *Middle-Impl*
imports *Abstract-Impl Pointer-Map*
begin

For the lack of a better name, the suffix mi stands for middle-implementation. This relects that this "implementation" is neither entirely abstract, nor has it been made fully concrete: the data structures are decided, but not their implementations.

record *bdd* =
  dpm :: (*nat* × *nat* × *nat*) *pointermap*
  dcl :: ((*nat* × *nat* × *nat*),*nat*) *map*

definition *emptymi* ≡ (|*dpm* = *empty-pointermap*, *dcl* = *Map.empty*|)

fun *destrmi* :: *nat* ⇒ *bdd* ⇒ (*nat*, *nat*) *IFEXD* **where**
destrmi 0 bdd = *FD* |
destrmi (*Suc 0*) *bdd* = *TD* |
destrmi (*Suc* (*Suc n*)) *bdd* = (*case pm-pth* (*dpm bdd*) *n of* (*v*, *t*, *e*) ⇒ *IFD v t e*)
fun *tmi* **where** *tmi bdd* = (*1*, *bdd*)
fun *fmi* **where** *fmi bdd* = (*0*, *bdd*)
fun *ifmi* :: *nat* ⇒ *nat* ⇒ *nat* ⇒ *bdd* ⇒ (*nat* × *bdd*) **where**
ifmi v t e bdd = (*if t* = *e*
  then (*t*, *bdd*)
  else (*let* (*r*,*pm*) = *pointermap-getmk* (*v*, *t*, *e*) (*dpm bdd*) *in*
  (*Suc* (*Suc r*), *dpm-update* (*const pm*) *bdd*)))

fun *Rmi-g* :: *nat* ⇒ *nat ifex* ⇒ *bdd* ⇒ *bool* **where**
Rmi-g 0 Falseif bdd = *True* |
Rmi-g (*Suc 0*) *Trueif bdd* = *True* |
Rmi-g (*Suc* (*Suc n*)) (*IF v t e*) *bdd* = (*pointermap-p-valid n* (*dpm bdd*)
  ∧ (*case pm-pth* (*dpm bdd*) *n of* (*nv*, *nt*, *ne*) ⇒ *nv* = *v* ∧ *Rmi-g nt t bdd* ∧ *Rmi-g ne e bdd*)) |
Rmi-g - - - = *False*

definition *Rmi s* ≡ {(*a*,*b*)|*a b*. *Rmi-g a b s*}

interpretation *mi-pre*: *bdd-impl-cmp-pre Rmi* **by** −

definition *bdd-node-valid bdd n* ≡ *n* ∈ *Domain* (*Rmi bdd*)
lemma [*simp*]:
  bdd-node-valid bdd 0
  bdd-node-valid bdd (*Suc 0*)
  apply(*simp-all add*: *bdd-node-valid-def Rmi-def*)
   using *Rmi-g.simps*(*1*,*2*) **apply** *blast*+
  done

definition *ifexd-valid bdd e* ≡ (*case e of IFD - t e* ⇒ *bdd-node-valid bdd t* ∧ *bdd-node-valid bdd e* | - ⇒ *True*)

definition *bdd-sane bdd* ≡ *pointermap-sane* (*dpm bdd*) ∧ *mi-pre.map-invar-impl* (*dcl bdd*) *bdd*

lemma [*simp*,*intro*!]: *bdd-sane emptymi*
  unfolding *emptymi-def bdd-sane-def bdd.simps*
by(*simp add*: *mi-pre.map-invar-impl-def*)

lemma *prod-split3*: *P* (*case p of* (*x*, *xa*, *xaa*) ⇒ *f x xa xaa*) = (∀ *x1 x2 x3*. *p* =

(*x1*, *x2*, *x3*) ⟶ *P* (*f x1 x2 x3*))
by(*simp split*: *prod.splits*)

lemma *IfI*: (*c* ⟹ *P x*) ⟹ (¬*c* ⟹ *P y*) ⟹ *P* (*if c then x else y*) **by** *simp*
lemma *fstsndI*: *x* = (*a*,*b*) ⟹ *fst x* = *a* ∧ *snd x* = *b* **by** *simp*
thm *nat.split*
lemma *Rmi-g-2-split*: *P* (*Rmi-g n x m*) =
  ((*x* = *Falseif* ⟶ *P* (*Rmi-g n x m*)) ∧
  (*x* = *Trueif* ⟶ *P* (*Rmi-g n x m*)) ∧
  (∀ *vs ts es*. *x* = *IF vs ts es* ⟶ *P* (*Rmi-g n x m*)))
by(*cases x*;*simp*)

lemma *rmigeq*: *Rmi-g ni1 n1 s* ⟹ *Rmi-g ni2 n2 s* ⟹ *ni1* = *ni2* ⟹ *n1* = *n2*
proof(*induction ni1 n1 s arbitrary*: *n2 ni2 rule*: *Rmi-g.induct*, *goal-cases*)
  case (*3 n v t e bdd n2 ni2*) **note** *goal3* = *3*
  note *1* = *goal3*(*1*,*2*)
  have *2*: *Rmi-g* (*fst* (*snd* (*pm-pth* (*dpm bdd*) *n*))) *t bdd Rmi-g* (*snd* (*snd* (*pm-pth* (*dpm bdd*) *n*))) *e bdd* **using** *goal3*(*3*) **by**(*clarsimp*)+
  note *mIH* = *1*(*1*)[*OF* - - *2*(*1*) - *refl*] *1*(*2*)[*OF* - - *2*(*2*) - *refl*]
  obtain *v2 t2 e2* **where** *v2*: *n2* = *IF v2 t2 e2* **using** *Rmi-g.simps*(*4*,*6*) *goal3*(*3*−*5*) **by**(*cases n2*) *blast*+
  thus *?case* **using** *goal3*(*3*−*4*) **by**(*clarsimp simp add*: *v2 goal3*(*5*)[*symmetric*] *mIH*)
qed (*rename-tac n2 ni2*, (*case-tac n2*; *clarsimp*))+

lemma *rmigneq*: *bdd-sane s* ⟹ *Rmi-g ni1 n1 s* ⟹ *Rmi-g ni2 n2 s* ⟹ *ni1* ≠ *ni2* ⟹ *n1* ≠ *n2*
proof(*induction ni1 n1 s arbitrary*: *n2 ni2 rule*: *Rmi-g.induct*, *goal-cases*)
  case *1* **thus** *?case* **by** (*metis Rmi-g.simps*(*6*) *old.nat.exhaust*)
next
  case *2* **thus** *?case* **by** (*metis Rmi-g.simps*(*4*,*8*) *old.nat.exhaust*)
next
  case (*3 n v t e bdd n2 ni2*) **note** *goal3* = *3*
  let *?bddpth* = *pm-pth* (*dpm bdd*)
  note *1* = *goal3*(*1*,*2*)[*OF prod.collapse prod.collapse*]
  have *2*: *Rmi-g* (*fst* (*snd* (*?bddpth n*))) *t bdd Rmi-g* (*snd* (*snd* (*?bddpth n*))) *e bdd* **using** *goal3*(*4*) **by**(*clarsimp*)+
  note *mIH* = *1*(*1*)[*OF goal3*(*3*) *2*(*1*)] *1*(*2*)[*OF goal3*(*3*) *2*(*2*)]
  show *?case* **proof**(*cases 0* < *ni2*, *case-tac 1* < *ni2*)
    case *False*
    hence *e*: *ni2* = *0* **by** *simp*
    with *goal3*(*5*) **have** *n2* = *Falseif* **using** *rmigeq* **by** *auto*
    thus *?thesis* **by** *simp*
  next
    case *True* **moreover assume** *3*: ¬ *1* < *ni2*
    ultimately have *ni2* = *1* **by** *simp*
    with *goal3*(*5*) **have** *n2* = *Trueif* **using** *rmigeq* **by** *auto*
    thus *?thesis* **by** *simp*
  next

assume *3*: *1* < *ni2*
then obtain *ni2s* **where** [*simp*]: *ni2* = *Suc* (*Suc ni2s*) **unfolding** *One-nat-def* **using** *less-imp-Suc-add* **by** *blast*
obtain *v2 t2 e2* **where** *v2*[*simp*]: *n2* = *IF v2 t2 e2* **using** *goal3*(*5*) **by**(*cases* (*ni2*, *n2*, *bdd*) *rule*: *Rmi-g.cases*) *clarsimp+*
have *4*: *Rmi-g* (*fst* (*snd* (*?bddpth ni2s*))) *t2 bdd Rmi-g* (*snd* (*snd* (*?bddpth ni2s*))) *e2 bdd* **using** *goal3*(*5*) **by** *clarsimp+*
show *?case* **unfolding** *v2*
proof(*cases fst* (*snd* (*?bddpth n*)) = *fst* (*snd* (*?bddpth ni2s*)),
case-tac snd (*snd* (*?bddpth n*)) = *snd* (*snd* (*?bddpth ni2s*)),
case-tac v = *v2*)
have *ne*: *ni2s* ≠ *n* **using** *goal3*(*6*) **by** *simp*
have *ib*: *pointermap-p-valid n* (*dpm bdd*) *pointermap-p-valid ni2s* (*dpm bdd*) **using** *Rmi-g.simps*(*3*) *goal3*(*4*,*5*) **by** *simp-all*
assume *goal1*:
fst (*snd* (*pm-pth* (*dpm bdd*) *n*)) = *fst* (*snd* (*pm-pth* (*dpm bdd*) *ni2s*))
snd (*snd* (*pm-pth* (*dpm bdd*) *n*)) = *snd* (*snd* (*pm-pth* (*dpm bdd*) *ni2s*))
v = *v2*
hence *?bddpth n* = *?bddpth ni2s* **unfolding** *prod-eq-iff* **using** *goal3*(*4*) *goal3*(*5*) **by** *auto*
with *goal3*(*3*) *ne* **have** *False* **unfolding** *bdd-sane-def* **using** *pth-eq-iff-index-eq*[*OF* - *ib*] **by** *simp*
thus *IF v t e* ≠ *IF v2 t2 e2* **..**
qed (*simp-all add*: *mIH*(*1*)[*OF 4*(*1*)] *mIH*(*2*)[*OF 4*(*2*)])
qed
qed *simp-all*

lemma *ifmi-les-hlp*: *pointermap-sane* (*dpm s*) ⟹ *pointermap-getmk* (*v*, *ni1*, *ni2*) (*dpm s*) = (*x1*, *dpm s′*) ⟹ *Rmi-g nia n s* ⟹ *Rmi-g nia n s′*
proof(*induction nia n s rule*: *Rmi-g.induct*, *goal-cases*)
case (*3 n v t e bdd*) **note** *goal3* = *3*
obtain *x1a x2a* **where** *pth*[*simp*]: *pm-pth* (*dpm bdd*) *n* = (*v*, *x1a*, *x2a*) **using** *goal3*(*5*) **by** *force*
have *pth′*[*simp*]: *pm-pth* (*dpm s′*) *n* = (*v*, *x1a*, *x2a*) **unfolding** *pth*[*symmetric*] **using** *goal3*(*4*,*5*) **by** (*meson Rmi-g.simps*(*3*) *pointermap-p-pth-inv*)
note *mIH* = *goal3*(*1*,*2*)[*OF pth*[*symmetric*] *refl goal3*(*3*,*4*)]
from *goal3*(*5*) **show** *?case*
unfolding *Rmi-g.simps*
using *pointermap-p-valid-inv*[*OF* - *goal3*(*4*)] *mIH*
by(*simp split*: *prod.splits*)
qed *simp-all*
lemma *ifmi-les*:
assumes *bdd-sane s*
assumes *ifmi v ni1 ni2 s* = (*ni*, *s′*)
shows *mi-pre.les s s′*
using *assms*
by(*clarsimp simp*: *bdd-sane-def comp-def apfst-def map-prod-def mi-pre.les-def Rmi-def ifmi-les-hlp split*: *if-splits prod.splits*)

lemma *ifmi-notouch-dcl*: *ifmi v ni1 ni2 s* = (*ni*, *s'*) ⟹ *dcl s'* = *dcl s*
by(*clarsimp split*: *if-splits prod.splits*)

lemma *ifmi-saneI*: *bdd-sane s* ⟹ *ifmi v ni1 ni2 s* = (*ni*, *s'*) ⟹ *bdd-sane s'*
apply(*subst bdd-sane-def*)
apply(*rule conjI*)
apply(*clarsimp simp*: *comp-def apfst-def map-prod-def bdd-sane-def split*: *if-splits option.splits split*: *prod.splits*)
apply(*rule conjunct1*[*OF pointermap-sane-getmkD*, *of dpm s* (*v*, *ni1*, *ni2*) -])
apply(*simp-all*)[*2*]
apply(*frule* (*1*) *ifmi-les*)
apply(*unfold bdd-sane-def*, *clarify*)
apply(*rule mi-pre.map-invar-impl-les*[*rotated*])
apply *assumption*
apply(*drule ifmi-notouch-dcl*)
apply(*simp*)
done

lemma *rmigif*: *Rmi-g ni* (*IF v n1 n2*) *s* ⟹ ∃ *n*. *ni* = *Suc* (*Suc n*)
apply(*cases ni*)
apply(*simp split*: *if-splits prod.splits*)
apply(*rename-tac nis*)
apply(*case-tac nis*)
apply(*simp split*: *if-splits prod.splits*)
apply(*simp split*: *if-splits prod.splits*)
done

lemma *in-lesI*:
assumes *mi-pre.les s s'*
assumes (*ni1*, *n1*) ∈ *Rmi s*
assumes (*ni2*, *n2*) ∈ *Rmi s*
shows (*ni1*, *n1*) ∈ *Rmi s'* (*ni2*, *n2*) ∈ *Rmi s'*
by (*meson assms mi-pre.les-def*)+

lemma *ifmi-modification-validI*:
assumes *sane*: *bdd-sane s*
assumes *ifm*: *ifmi v ni1 ni2 s* = (*ni*, *s'*)
assumes *vld*: *bdd-node-valid s n*
shows *bdd-node-valid s' n*
proof(*cases ni1* = *ni2*)
case *True* **with** *ifm vld* **show** *?thesis* **by** *simp*
next
case *False*
{
fix *b*
from *ifm* **have** (*n*, *b*) ∈ *Rmi s* ⟹ (*n*, *b*) ∈ *Rmi s'*
by(*induction n b* - *rule*: *Rmi-g.induct*) (*auto dest*: *pointermap-p-pth-inv pointermap-p-valid-inv simp*: *apfst-def map-prod-def False Rmi-def split*: *prod.splits*)

}
thus *?thesis*
using *vld* **unfolding** *bdd-node-valid-def* **by** *blast*
qed

definition *tmi′ s ≡ do {oassert (bdd-sane s); Some (tmi s)}*
definition *fmi′ s ≡ do {oassert (bdd-sane s); Some (fmi s)}*
definition *ifmi′ v ni1 ni2 s ≡ do {oassert (bdd-sane s ∧ bdd-node-valid s ni1 ∧ bdd-node-valid s ni2); Some (ifmi v ni1 ni2 s)}*

lemma *ifmi′-spec*: ⟦*bdd-sane s*; *bdd-node-valid s ni1*; *bdd-node-valid s ni2*⟧ ⟹ *ospec (ifmi′ v ni1 ni2 s) (λr. r = ifmi v ni1 ni2 s)*
unfolding *ifmi′-def* **by**(*simp split*: *Option.bind-splits*)
lemma *ifmi′-ifmi*: ⟦*bdd-sane s*; *bdd-node-valid s ni1*; *bdd-node-valid s ni2*⟧ ⟹ *ifmi′ v ni1 ni2 s = Some (ifmi v ni1 ni2 s)*
unfolding *ifmi′-def* **by**(*simp split*: *Option.bind-splits*)

definition *destrmi′ ni s ≡ do {oassert (bdd-sane s ∧ bdd-node-valid s ni); Some (destrmi ni s)}*

lemma *destrmi-someD*: *destrmi′ e bdd = Some x ⟹ bdd-sane bdd ∧ bdd-node-valid bdd e*
by(*simp add*: *destrmi′-def split*: *Option.bind-splits*)

lemma *Rmi-sv*:
assumes *bdd-sane s (ni,n) ∈ Rmi s (ni′,n′) ∈ Rmi s*
shows *ni=ni′ ⟹ n=n′*
and *ni≠ni′ ⟹ n≠n′*
using *assms*
apply *safe*
apply (*simp-all add*: *Rmi-def*)
using *rmigeq* **apply** *simp*
apply (*drule (3) rmigneq*)
by *clarify*

lemma *True-rep*[*simp*]: *bdd-sane s ⟹ (ni,Trueif)∈Rmi s ⟷ ni=Suc 0*
using *Rmi-def Rmi-g.simps(2) Rmi-sv(2)* **by** *blast*

lemma *False-rep*[*simp*]: *bdd-sane s ⟹ (ni,Falseif)∈Rmi s ⟷ ni=0*
using *Rmi-def Rmi-g.simps(1) Rmi-sv(2)* **by** *blast*

definition *updS s x r = dcl-update (λm. m(x ↦ r)) s*
thm *Rmi-g.induct*

lemma *updS-dpm*: *dpm (updS s x r) = dpm s*
unfolding *updS-def* **by** *simp*

lemma *updS-Rmi-g*: *Rmi-g n i (updS s x r) = Rmi-g n i s*
apply(*induction n i s rule*: *Rmi-g.induct*)

apply(*simp-all*) **unfolding** *updS-dpm* **by** *auto*

lemma *updS-Rmi*: *Rmi* (*updS s x r*) = *Rmi s*
unfolding *Rmi-def updS-Rmi-g* **by** *blast*

interpretation *mi*: *bdd-impl-cmp bdd-sane Rmi tmi′ fmi′ ifmi′ destrmi′ dcl updS* (=)
proof −
note *s* = *mi-pre.les-def*[*simp*] *Rmi-def*

note [*simp*] = *tmi′-def fmi′-def ifmi′-def destrmi′-def apfst-def map-prod-def*

show *bdd-impl-cmp bdd-sane Rmi tmi′ fmi′ ifmi′ destrmi′ dcl updS* (=)
proof(*unfold-locales*, *goal-cases*)
case *1* **thus** *?case* **by**(*clarsimp split*: *if-splits simp*: *Rmi-def*)
next case *2* **thus** *?case* **by**(*clarsimp split*: *if-splits simp*: *Rmi-def*)
next case (*3 s ni1 n1 ni2 n2 v*) **note** *goal3* = *3*
note [*simp*] = *Rmi-sv*[*OF this*]
have *e*: *n1* = *n2* ⟹ *ni1* = *ni2* **by**(*rule ccontr*) *simp*
obtain *ni s′* **where**[*simp*]: (*ifmi′ v ni1 ni2 s*) = *Some* (*ni*, *s′*)
unfolding *ifmi′-def bdd-node-valid-def* **using** *goal3* **by**(*simp add*: *DomainI del*: *ifmi.simps*) *fastforce*
hence *ifm*: *ifmi v ni1 ni2 s* = (*ni*, *s′*)
using *goal3* **unfolding** *ifmi′-def bdd-node-valid-def*
by(*simp add*: *DomainI*)
have *ifmi′-ospec*: ⋀*P*. *ospec* (*ifmi′ v ni1 ni2 s*) *P* ⟷ *P* (*ifmi v ni1 ni2 s*) **by**(*simp del*: *ifmi′-def ifmi.simps add*: *ifm*)
from *goal3* **show** *?case*
unfolding *ifmi′-ospec*
apply(*split prod.splits*; *clarify*)
apply(*rule conjI*)

apply(*clarsimp simp*: *Rmi-def IFC-def bdd-sane-def ifmi-les-hlp pointermap-sane-getmkD pointermap-update-pthI split*: *if-splits prod.splits*)

using *ifmi-les*[*OF* ‹*bdd-sane s*› *ifm*] *ifmi-saneI*[*OF* ‹*bdd-sane s*› *ifm*] *ifm* **apply**(*simp*)
done
next case *4* **thus** *?case*
apply (*clarsimp split*: *Option.bind-splits if-splits*)
done
next case *5* **thus** *?case* **by**(*clarsimp split*: *if-splits*)
next case *6* **thus** *?case*
apply (*clarsimp simp add*: *bdd-node-valid-def split*: *Option.bind-splits if-splits*)
apply (*auto simp*: *Rmi-def elim*: *Rmi-g.elims*)
done
next
case *7* **thus** *?case* **using** *Rmi-sv* **by** *blast*

next
case *8* **thus** *?case* **using** *Rmi-sv* **by** *blast*
next
case *9* **thus** *?case* **unfolding** *bdd-sane-def* **by** *simp*
next
case *10* **thus** *?case* **unfolding** *bdd-sane-def mi-pre.map-invar-impl-def* **using** *updS-Rmi*
by(*clarsimp simp add*: *updS-def simp del*: *ifex-ite-opt.simps*) *blast*
next
case *11* **thus** *?case* **using** *updS-Rmi* **by** *auto*
qed
qed

lemma *p-valid-RmiI*: (*Suc* (*Suc na*), *b*) ∈ *Rmi bdd* ⟹ *pointermap-p-valid na* (*dpm bdd*)
unfolding *Rmi-def* **by**(*cases b*) (*auto*)
lemma *n-valid-RmiI*: (*na*, *b*) ∈ *Rmi bdd* ⟹ *bdd-node-valid bdd na*
unfolding *bdd-node-valid-def*
by(*intro DomainI*, *assumption*)
lemma *n-valid-Rmi-alt*: *bdd-node-valid bdd na* ⟷ (∃ *b*. (*na*, *b*) ∈ *Rmi bdd*)
unfolding *bdd-node-valid-def*
by *auto*

lemma *ifmi-result-validI*:
assumes *sane*: *bdd-sane s*
assumes *vld*: *bdd-node-valid s ni1 bdd-node-valid s ni2*
assumes *ifm*: *ifmi v ni1 ni2 s* = (*ni*, *s′*)
shows *bdd-node-valid s′ ni*
proof −
from *vld* **obtain** *n1 n2* **where** (*ni1*, *n1*) ∈ *Rmi s* (*ni2*, *n2*) ∈ *Rmi s* **unfolding** *bdd-node-valid-def* **by** *blast*
note *mi.IFimpl-rule*[*OF sane this*]
note *this*[*unfolded ifmi′-ifmi*[*OF sane vld*] *ospec.simps*, *of v*, *unfolded ifm*, *unfolded prod.simps*]
thus *?thesis* **unfolding** *bdd-node-valid-def* **by** *blast*
qed

end

8 Array List

Most of this has been contributed by Peter Lammich.

theory *Array-List*
imports
Separation-Logic-Imperative-HOL.Array-Blit
begin

This implements a datastructure that efficiently supports two operations: appending an element and looking up the nth element. The implementation is straightforward.

As underlying data structure an array is used. Since changing the length of an array requires copying, we double the size whenever the array needs to be expanded. We use a counter for the current length to track which elements are used and which are spares.

type-synonym *'a array-list = 'a array × nat*

definition *is-array-list l* ≡ $\lambda(a,n).\ \exists_A l'.\ a \mapsto_a l' * \uparrow(n \leq length\ l' \wedge l = take\ n\ l' \wedge length\ l'>0)$

definition *initial-capacity* ≡ *16::nat*

definition *arl-empty* ≡ *do* {
  a ← *Array.new initial-capacity default*;
  return (*a*,*0*)
}

lemma [*sep-heap-rules*]: < *emp* > *arl-empty* <*is-array-list* []>
  by (*sep-auto simp*: *arl-empty-def is-array-list-def initial-capacity-def*)

definition *arl-nth* ≡ λ(*a*,*n*) *i*. *do* {
  Array.nth a i
}

lemma [*sep-heap-rules*]: *i*<*length l* ⟹ < *is-array-list l a* > *arl-nth a i* <λ*x*. *is-array-list l a* ∗ ↑(*x* = *l*!*i*) >
  by (*sep-auto simp*: *arl-nth-def is-array-list-def split*: *prod.splits*)

definition *arl-append* ≡ λ(*a*,*n*) *x*. *do* {
  len ← *Array.len a*;

  if n<*len then do* {
    a ← *Array.upd n x a*;
    return (*a*,*n*+*1*)
  } *else do* {
    let newcap = *2* ∗ *len*;
    a ← *array-grow a newcap default*;
    a ← *Array.upd n x a*;
    return (*a*,*n*+*1*)
  }
}

lemma [*sep-heap-rules*]:
  < *is-array-list l a* >
    arl-append a x
  <λ*a*. *is-array-list* (*l*@[*x*]) *a* >$_t$

by (*sep-auto*
simp: *arl-append-def is-array-list-def take-update-last neq-Nil-conv*
split: *prod.splits nat.split*)

lemma *is-array-list-prec*: *precise is-array-list*
unfolding *is-array-list-def*[*abs-def*]
apply(*rule preciseI*)
apply(*simp split*: *prod.splits*)
using *preciseD snga-prec* **by** *fastforce*

lemma *is-array-list-lengthIA*: *is-array-list l li* $\Longrightarrow_A$ $\uparrow$(*snd li* = *length l*) $*$ *true*
by(*sep-auto simp*: *is-array-list-def split*: *prod.splits*)
find-consts *assn* $\Rightarrow$ *bool*
lemma *is-array-list-lengthI*: *x* $\models$ *is-array-list l li* $\Longrightarrow$ *snd li* = *length l*
using *is-array-list-lengthIA* **by** (*metis* (*full-types*) *ent-pure-post-iff star-aci*(*2*))

end

9 Imparative implementation for Pointermap

theory *Pointer-Map-Impl*
imports *Array-List*
Separation-Logic-Imperative-HOL.Sep-Main
Separation-Logic-Imperative-HOL.Hash-Map-Impl
Pointer-Map
begin

record *'a pointermap-impl* =
entriesi :: *'a array-list*
getentryi :: (*'a*,*nat*) *hashtable*
lemma *pointermapieq-exhaust*: *entries a* = *entries b* $\Longrightarrow$ *getentry a* = *getentry b* $\Longrightarrow$ *a* = (*b* :: *'a pointermap*) **by** *simp*

definition *is-pointermap-impl* :: (*'a*::{*hashable*,*heap*}) *pointermap* $\Rightarrow$ *'a pointermap-impl* $\Rightarrow$ *assn* **where**
is-pointermap-impl b bi $\equiv$
is-array-list (*entries b*) (*entriesi bi*)
$*$ *is-hashmap* (*getentry b*) (*getentryi bi*)

lemma *is-pointermap-impl-prec*: *precise is-pointermap-impl*
unfolding *is-pointermap-impl-def*[*abs-def*]
apply(*rule preciseI*)
apply(*clarsimp*)
apply(*rename-tac a a' x y p F F'*)
apply(*rule pointermapieq-exhaust*)
apply(*rule-tac p* = *entriesi p* **and** *h* = (*x*,*y*) **in** *preciseD*[*OF is-array-list-prec*])
apply(*unfold star-aci*(*1*))
apply *blast*
apply(*rule-tac p* = *getentryi p* **and** *h* = (*x*,*y*) **in** *preciseD*[*OF is-hashmap-prec*])

```
  apply(simp only: star-aci(2)[symmetric])
  apply(simp only: star-aci(1)[symmetric])
  apply(simp only: star-aci(2)[symmetric])
  done

  definition pointermap-empty where
    pointermap-empty ≡ do {
      hm ← hm-new;
      arl ← arl-empty;
      return (|entriesi = arl, getentryi = hm |)
    }

 lemma [sep-heap-rules]: < emp > pointermap-empty <is-pointermap-impl empty-pointermap>t
    unfolding is-pointermap-impl-def
    by (sep-auto simp: pointermap-empty-def empty-pointermap-def)

  definition pm-pthi where
    pm-pthi m p ≡ arl-nth (entriesi m) p

  lemma [sep-heap-rules]: pointermap-sane m ⟹ pointermap-p-valid p m ⟹
    < is-pointermap-impl m mi > pm-pthi mi p <λai. is-pointermap-impl m mi *
↑(ai = pm-pth m p)>
   by (sep-auto simp: pm-pthi-def pm-pth-def is-pointermap-impl-def pointermap-p-valid-def)

  definition pointermap-getmki where
    pointermap-getmki a m ≡ do {
        lo ← ht-lookup a (getentryi m);
        (case lo of
          Some l ⇒ return (l,m) |
          None ⇒ do {
            p ← return (snd (entriesi m));
        ent ← arl-append (entriesi m) a;
        lut ← hm-update a p (getentryi m);
        u ← return (|entriesi = ent, getentryi = lut|);
        return (p,u)
          }
        )
    }

  lemmas pointermap-getmki-defs = pointermap-getmki-def pointermap-getmk-def
pointermap-insert-def is-pointermap-impl-def
  lemma [sep-heap-rules]: pointermap-sane m ⟹ pointermap-getmk a m = (p,u)
⟹
    < is-pointermap-impl m mi >
      pointermap-getmki a mi
    <λ(pi,ui). is-pointermap-impl u ui * ↑(pi = p)>t
  apply(cases getentry m a)
   apply(unfold pointermap-getmki-def)
   apply(unfold return-bind)
```

apply(*rule bind-rule*[**where** $R = \lambda r.$ *is-pointermap-impl m mi* $*$ $\uparrow$(r = *None* $\wedge$ (*snd* (*entriesi mi*) = *p*)) $*$ *true*])
apply(*sep-auto simp*: *pointermap-getmki-defs is-array-list-def split*: *prod.splits*;*fail*)
apply(*sep-auto simp*: *pointermap-getmki-defs*)+
done

end

10 Imparative implementation

theory *Conc-Impl*
imports *Pointer-Map-Impl Middle-Impl*
begin

record *bddi* =
dpmi :: (*nat* × *nat* × *nat*) *pointermap-impl*
dcli :: ((*nat* × *nat* × *nat*),*nat*) *hashtable*
lemma *bdd-exhaust*: *dpm a* = *dpm b* $\Longrightarrow$ *dcl a* = *dcl b* $\Longrightarrow$ *a* = (*b* :: *bdd*) **by** *simp*

instantiation *prod* :: (*default*, *default*) *default*
begin
definition *default-prod* :: ('*a* × '*b*) ≡ (*default*, *default*)
instance ..
end

instantiation *nat* :: *default*
begin
definition *default-nat* ≡ *0* :: *nat*
instance ..
end

definition *is-bdd-impl* (*bdd*::*bdd*) (*bddi*::*bddi*) = *is-pointermap-impl* (*dpm bdd*) (*dpmi bddi*) $*$ *is-hashmap* (*dcl bdd*) (*dcli bddi*)

lemma *is-bdd-impl-prec*: *precise is-bdd-impl*
apply(*rule preciseI*)
apply(*unfold is-bdd-impl-def*)
apply(*clarsimp*)
apply(*rename-tac a a' x y p F F'*)
apply(*rule bdd-exhaust*)
apply(*rule-tac p* = *dpmi p* **and** *h* = (*x*,*y*) **in** *preciseD*[*OF is-pointermap-impl-prec*])
apply(*unfold star-aci*(*1*))
apply *blast*
apply(*rule-tac p* = *dcli p* **and** *h* = (*x*,*y*) **in** *preciseD*[*OF is-hashmap-prec*])
apply(*simp only*: *star-aci*(*2*)[*symmetric*])
apply(*simp only*: *star-aci*(*1*)[*symmetric*])
apply(*simp only*: *star-aci*(*2*)[*symmetric*])

done

```
definition emptyci :: bddi Heap ≡ do { ep ← pointermap-empty; ehm ← hm-new;
return (|dpmi=ep, dcli=ehm|) }
definition tci bdd ≡ return (1::nat,bdd::bddi)
definition fci bdd ≡ return (0::nat,bdd::bddi)
definition ifci v t e bdd ≡ (if t = e then return (t, bdd) else do {
                              (p,u) ← pointermap-getmki (v, t, e) (dpmi bdd);
                              return (Suc (Suc p), dpmi-update (const u) bdd)
})
definition destrci :: nat ⇒ bddi ⇒ (nat, nat) IFEXD Heap where
destrci n bdd ≡ (case n of
  0 ⇒ return FD |
  Suc 0 ⇒ return TD |
  Suc (Suc p) ⇒ pm-pthi (dpmi bdd) p ≫= (λ(v,t,e). return (IFD v t e)))

term mi.les

lemma emptyci-rule[sep-heap-rules]: <emp> emptyci <is-bdd-impl emptymi>_t
  by(sep-auto simp: is-bdd-impl-def emptyci-def emptymi-def)

lemma [sep-heap-rules]: tmi′ bdd = Some (p,bdd′)
  ⟹ <is-bdd-impl bdd bddi>
       tci bddi
     <λ(pi,bddi′). is-bdd-impl bdd′ bddi′ ∗ ↑(pi = p)>
  by (sep-auto simp: tci-def tmi′-def split: Option.bind-splits)

lemma [sep-heap-rules]: fmi′ bdd = Some (p,bdd′)
  ⟹ <is-bdd-impl bdd bddi>
       fci bddi
     <λ(pi,bddi′). is-bdd-impl bdd′ bddi′ ∗ ↑(pi = p)>
by(sep-auto simp: fci-def fmi′-def split: Option.bind-splits)

lemma [sep-heap-rules]: ifmi′ v t e bdd = Some (p, bdd′) ⟹
  <is-bdd-impl bdd bddi> ifci v t e bddi
  <λ(pi,bddi′). is-bdd-impl bdd′ bddi′ ∗ ↑(pi = p)>_t
  apply(clarsimp simp: is-bdd-impl-def ifmi′-def simp del: ifmi.simps)
  by (sep-auto simp: ifci-def apfst-def map-prod-def is-bdd-impl-def bdd-sane-def
             split: prod.splits if-splits Option.bind-splits)

lemma destrci-rule[sep-heap-rules]:
  destrmi′ n bdd = Some r ⟹
  <is-bdd-impl bdd bddi> destrci n bddi
  <λr′. is-bdd-impl bdd bddi ∗ ↑(r′ = r)>
  unfolding destrmi′-def apply (clarsimp split: Option.bind-splits)
  apply(cases (n, bdd) rule: destrmi.cases)
    by (sep-auto simp: destrci-def bdd-node-valid-def is-bdd-impl-def ifexd-valid-def
bdd-sane-def
                 dest: p-valid-RmiI)+
```

term *mi.restrict-top-impl*

thm *mi.case-ifexi-def*

definition *case-ifexici fti ffi fii ni bddi* ≡ *do* {
  dest ← *destrci ni bddi*;
  case dest of TD ⇒ *fti* | *FD* ⇒ *ffi* | *IFD v ti ei* ⇒ *fii v ti ei*
}

lemma [*sep-decon-rules*]:
  assumes *S*: *mi.case-ifexi fti ffi fii ni bdd* = *Some r*
  assumes [*sep-heap-rules*]:
    destrmi′ ni bdd = *Some TD* ⟹ *fti bdd* = *Some r* ⟹ <*is-bdd-impl bdd bddi*> *ftci* <*Q*>
    destrmi′ ni bdd = *Some FD* ⟹ *ffi bdd* = *Some r* ⟹ <*is-bdd-impl bdd bddi*> *ffci* <*Q*>
    ⋀*v t e*. *destrmi′ ni bdd* = *Some* (*IFD v t e*) ⟹ *fii v t e bdd* = *Some r*
     ⟹ <*is-bdd-impl bdd bddi*> *fici v t e* <*Q*>
  shows <*is-bdd-impl bdd bddi*> *case-ifexici ftci ffci fici ni bddi* <*Q*>
  using *S* **unfolding** *mi.case-ifexi-def* **apply** (*clarsimp split*: *Option.bind-splits IFEXD.splits*)
  by (*sep-auto simp*: *case-ifexici-def*)+

definition *restrict-topci p vr vl bdd* =
  case-ifexici
    (*return p*)
    (*return p*)
    (*λv te ee*. *return* (*if v* = *vr then* (*if vl then te else ee*) *else p*))
    p bdd

lemma [*sep-heap-rules*]:
  assumes *mi.restrict-top-impl p var val bdd* = *Some* (*r*,*bdd′*)
  shows <*is-bdd-impl bdd bddi*> *restrict-topci p var val bddi*
        <*λri*. *is-bdd-impl bdd bddi* ∗ ↑(*ri* = *r*)>
  using *assms* **unfolding** *mi.restrict-top-impl-def restrict-topci-def* **by** *sep-auto*

fun *lowest-topsci* **where**
lowest-topsci [] *s* = *return None* |
lowest-topsci (*e*#*es*) *s* =
    case-ifexici
      (*lowest-topsci es s*)
      (*lowest-topsci es s*)
      (*λv t e*. *do* {
      (*rec*) ← *lowest-topsci es s*;
        (*case rec of*
          Some u ⇒ *return* ((*Some* (*min u v*))) |
          None ⇒ *return* ((*Some v*)))
       }) *e s*

declare *lowest-topsci.simps*[*simp del*]

lemma [*sep-heap-rules*]:
  assumes *mi.lowest-tops-impl es bdd = Some* (*r*,*bdd′*)
  shows <*is-bdd-impl bdd bddi*> *lowest-topsci es bddi*
  <λ(*ri*). *is-bdd-impl bdd bddi* ∗ ↑(*ri* = *r* ∧ *bdd′*=*bdd*)>
proof −
  note [*simp*] = *lowest-topsci.simps mi.lowest-tops-impl.simps*
  show *?thesis* **using** *assms*
    apply (*induction es arbitrary*: *bdd r bdd′ bddi*)
     apply (*sep-auto*)

    apply (*clarsimp simp*: *mi.case-ifexi-def split*: *Option.bind-splits IFEXD.splits*)
      apply (*sep-auto simp*: *mi.case-ifexi-def*)
     apply (*sep-auto simp*: *mi.case-ifexi-def*)
    apply (*sep-auto simp*: *mi.case-ifexi-def*)
    done
qed

partial-function(*heap*) *iteci* **where**
iteci i t e s = do {
  (*lt*) ← *lowest-topsci* [*i*, *t*, *e*] *s*;
  case lt of
    Some a ⇒ *do* {
      ti ← *restrict-topci i a True s*;
      tt ← *restrict-topci t a True s*;
      te ← *restrict-topci e a True s*;
      fi ← *restrict-topci i a False s*;
      ft ← *restrict-topci t a False s*;
      fe ← *restrict-topci e a False s*;
      (*tb*,*s′*) ← *iteci ti tt te s*;
      (*fb*,*s′′*) ← *iteci fi ft fe s′*;
      (*ifci a tb fb s′′*)
     }
  | *None* ⇒ *do* {
    case-ifexici (*return* (*t*,*s*)) (*return* (*e*,*s*)) (λ- - -. *raise STR ′′Cannot happen′′*) *i*
s
   }
  }
declare *iteci.simps*[*code*]

lemma *iteci-rule*:
  (*mi.ite-impl i t e bdd = Some* (*p*,*bdd′*)) ⟶
  <*is-bdd-impl bdd bddi*>
    iteci i t e bddi
  <λ(*pi*,*bddi′*). *is-bdd-impl bdd′ bddi′* ∗ ↑(*pi*=*p*)>$_t$
  apply (*induction arbitrary*: *i t e bddi bdd p bdd′ rule*: *mi.ite-impl.fixp-induct*)
    subgoal

```
      apply simp
      using option-admissible[where P=
            λ(((x1,x2),x3),x4) (r1,r2). ∀ bddi.
             <is-bdd-impl x4 bddi>
               iteci x1 x2 x3 bddi
             <λr. case r of (p_i, bddi') ⇒ is-bdd-impl r2 bddi' ∗ ↑ (p_i = r1)>_t]
      apply auto[1]
      apply (fo-rule subst[rotated])
       apply (assumption)
      by auto
    subgoal by simp
    subgoal
      apply clarify
      apply (clarsimp split: option.splits Option.bind-splits prod.splits)
       apply (subst iteci.simps)
       apply (sep-auto)
      apply (subst iteci.simps)
      apply (sep-auto)
       unfolding imp-to-meta apply rprems
       apply simp
      apply sep-auto
       apply (rule fi-rule)
        apply rprems
        apply simp
       apply frame-inference
      by sep-auto
  done

declare iteci-rule[THEN mp, sep-heap-rules]

definition param-optci where
  param-optci i t e bdd = do {
    (tr, bdd) ← tci bdd;
    (fl, bdd) ← fci bdd;
    id ← destrci i bdd;
    td ← destrci t bdd;
    ed ← destrci e bdd;
          return (
          if id = TD then Some t else
                  if id = FD then Some e else
                  if td = TD ∧ ed = FD then Some i else
                  if t = e then Some t else
                  if ed = TD ∧ i = t then Some tr else
                  if td = FD ∧ i = e then Some fl else
                  None, bdd)
  }

lemma param-optci-rule:
  ( mi.param-opt-impl i t e bdd = Some (p,bdd')) ⟹
```

```
  <is-bdd-impl bdd bddi>
    param-optci i t e bddi
  <λ(pi,bddi'). is-bdd-impl bdd' bddi' ∗ ↑(pi=p)>t
by (sep-auto simp add: mi.param-opt-impl.simps param-optci-def tmi'-def fmi'-def
            split: Option.bind-splits)

lemma bdd-hm-lookup-rule:
  (dcl bdd (i,t,e) = p) ⟹
  <is-bdd-impl bdd bddi>
    hm-lookup (i, t, e) (dcli bddi)
  <λ(pi). is-bdd-impl bdd bddi ∗ ↑(pi = p)>t
unfolding is-bdd-impl-def by (sep-auto)

lemma bdd-hm-update-rule'[sep-heap-rules]:
  <is-bdd-impl bdd bddi>
    hm-update k v (dcli bddi)
  <λr. is-bdd-impl (updS bdd k v) (dcli-update (const r) bddi) ∗ true>
unfolding is-bdd-impl-def updS-def by (sep-auto)

partial-function(heap) iteci-lu where
iteci-lu i t e s = do {
  lu ← ht-lookup (i,t,e) (dcli s);
  (case lu of Some b ⇒ return (b,s)
    | None ⇒ do {
      (po,s) ← param-optci i t e s;
      (case po of Some b ⇒ do {
        return (b,s)}
      | None ⇒ do {
        (lt) ← lowest-topsci [i, t, e] s;
        (case lt of Some a ⇒ do {
        ti ← restrict-topci i a True s;
        tt ← restrict-topci t a True s;
        te ← restrict-topci e a True s;
        fi ← restrict-topci i a False s;
        ft ← restrict-topci t a False s;
        fe ← restrict-topci e a False s;
        (tb,s) ← iteci-lu ti tt te s;
        (fb,s) ← iteci-lu fi ft fe s;
        (r,s) ← ifci a tb fb s;
        cl ← hm-update (i,t,e) r (dcli s);
        return (r,dcli-update (const cl) s)
       }
         | None ⇒ raise STR ''Cannot happen'' )})
  })}

term ht-lookup
declare iteci-lu.simps[code]
thm iteci-lu.simps[unfolded restrict-topci-def case-ifexici-def  param-optci-def low-
est-topsci.simps]
```

```
partial-function(heap) iteci-lu-code where iteci-lu-code i t e s = do {
  lu ← hm-lookup (i, t, e) (dcli s);
  case lu of None ⇒ let po = if i = 1 then Some t
                             else if i = 0 then Some e else if t = 1 ∧ e = 0 then Some
i else if t = e then Some t else if e = 1 ∧ i = t then Some 1 else if t = 0 ∧ i = e
then Some 0 else None
                   in case po of None ⇒ do {
                                  id ← destrci i s;
                                  td ← destrci t s;
                                  ed ← destrci e s;
                                  let a = (case id of IFD v t e ⇒ v);
                                  let a = (case td of IFD v t e ⇒ min a v | - ⇒ a);
                                  let a = (case ed of IFD v t e ⇒ min a v | - ⇒ a);
                                  let ti = (case id of IFD v ti ei ⇒ if v = a then ti
else i | - ⇒ i);
                                  let tt = (case td of IFD v ti ei ⇒ if v = a then ti
else t | - ⇒ t);
                                  let te = (case ed of IFD v ti ei ⇒ if v = a then ti
else e | - ⇒ e);
                                  let fi = (case id of IFD v ti ei ⇒ if v = a then ei
else i | - ⇒ i);
                                  let ft = (case td of IFD v ti ei ⇒ if v = a then ei
else t | - ⇒ t);
                                  let fe = (case ed of IFD v ti ei ⇒ if v = a then ei
else e | - ⇒ e);
                                  (tb, s) ← iteci-lu-code ti tt te s;
                                  (fb, s) ← iteci-lu-code fi ft fe s;
                                  (r, s) ← ifci a tb fb s;
                                  cl ← hm-update (i, t, e) r (dcli s);
                                  return (r, dcli-update (const cl) s)
                                }
                      | Some b ⇒ return (b, s)
  | Some b ⇒ return (b, s)
}

declare iteci-lu-code.simps[code]

lemma iteci-lu-code[code-unfold]: iteci-lu i t e s = iteci-lu-code i t e s
oops


lemma iteci-lu-rule:
  ( mi.ite-impl-lu i t e bdd = Some (p,bdd')) ⟶
  <is-bdd-impl bdd bddi>
    iteci-lu i t e bddi
  <λ(pi,bddi'). is-bdd-impl bdd' bddi' * ↑(pi=p )>_t
  apply (induction arbitrary: i t e bddi bdd p bdd' rule: mi.ite-impl-lu.fixp-induct)
    subgoal
      apply simp
```

```
  using option-admissible[where P=
        λ(((x1,x2),x3),x4) (r1,r2). ∀ bddi.
         <is-bdd-impl x4 bddi>
           iteci-lu x1 x2 x3 bddi
         <λr. case r of (p_i, bddi') ⇒ is-bdd-impl r2 bddi' ∗ ↑ (p_i = r1)>_t]
  apply auto[1]
  apply (fo-rule subst[rotated])
   apply (assumption)
  by auto
subgoal by simp
subgoal
  apply clarify
  apply (clarsimp split: option.splits Option.bind-splits prod.splits)
  subgoal
    unfolding updS-def
    apply (subst iteci-lu.simps)
    apply (sep-auto)
     using bdd-hm-lookup-rule apply(blast)
    apply(sep-auto)
     apply(rule fi-rule)
      apply(rule param-optci-rule)
      apply(sep-auto)
     apply(sep-auto)
    apply(sep-auto)
     unfolding imp-to-meta
     apply(rule fi-rule)
      apply(rprems)
      apply(simp; fail)
     apply(sep-auto)
    apply(sep-auto)
     apply(rule fi-rule)
      apply(rprems)
      apply(simp; fail)
     apply(sep-auto)
     apply(sep-auto)
    unfolding updS-def by (sep-auto)
  subgoal
    apply(subst iteci-lu.simps)
    apply(sep-auto)
     using bdd-hm-lookup-rule apply(blast)
    apply(sep-auto)
     apply(rule fi-rule)
      apply(rule param-optci-rule)
      apply(sep-auto)
     apply(sep-auto)
    by (sep-auto)
  subgoal
    apply(subst iteci-lu.simps)
    apply(sep-auto)
```

```
      using bdd-hm-lookup-rule apply(blast)
     by(sep-auto)
    done
  done
```

10.1 A standard library of functions

declare *iteci-rule*[*THEN mp*, *sep-heap-rules*]

```
definition notci e s ≡ do {
  (f,s) ← fci s;
  (t,s) ← tci s;
  iteci-lu e f t s
}
definition orci e1 e2 s ≡ do {
  (t,s) ← tci s;
  iteci-lu e1 t e2 s
}
definition andci e1 e2 s ≡ do {
  (f,s) ← fci s;
  iteci-lu e1 e2 f s
}
definition norci e1 e2 s ≡ do {
  (r,s) ← orci e1 e2 s;
  notci r s
}
definition nandci e1 e2 s ≡ do {
  (r,s) ← andci e1 e2 s;
  notci r s
}
definition biimpci a b s ≡ do {
  (nb,s) ← notci b s;
  iteci-lu a b nb s
}
definition xorci a b s ≡ do {
  (nb,s) ← notci b s;
  iteci-lu a nb b s
}
definition litci v bdd ≡ do {
  (t,bdd) ← tci bdd;
  (f,bdd) ← fci bdd;
  ifci v t f bdd
}
definition tautci v bdd ≡ do {
  d ← destrci v bdd;
  return (d = TD)
}
```

10.2 Printing

The following functions are exported unverified. They are intended for BDD debugging purposes.

partial-function(*heap*) *serializeci* :: *nat* ⇒ *bddi* ⇒ ((*nat* × *nat*) × *nat*) *list Heap*
where
serializeci p s = *do* {
  d ← *destrci p s*;
  (*case d of*
    IFD v t e ⇒ *do* {
      r ← *serializeci t s*;
      l ← *serializeci e s*;
      return (*remdups* ([((*p*,*t*),*1*),((*p*,*e*),*0*)] @ *r* @ *l*))
    } |
    - ⇒ *return* []
)
}
declare *serializeci.simps*[*code*]

fun *mapM* **where**
mapM f [] = *return* [] |
mapM f (*a*#*as*) = *do* {
  r ← *f a*;
  rs ← *mapM f as*;
  return (*r*#*rs*)
}
definition *liftM f ma* = *do* { *a* ← *ma*; *return* (*f a*) }
definition *sequence* = *mapM id*
term *liftM* (*map f*)
lemma *liftM* (*map f*) (*sequence l*) = *sequence* (*map* (*liftM f*) *l*)
  apply(*induction l*)
   apply(*simp add*: *sequence-def liftM-def*)
  apply(*simp*)
oops

fun *string-of-nat* :: *nat* ⇒ *string* **where**
  string-of-nat n = (*if n* < *10 then* [*char-of-nat* (*48* + *n*)]
                else string-of-nat (*n div 10*) @ [*char-of-nat* (*48* + (*n mod 10*))])

definition *labelci* :: *bddi* ⇒ *nat* ⇒ (*string* × *string* × *string*) *Heap* **where**
labelci s n = *do* {
   d ← *destrci n s*;
   let son = *string-of-nat n*;
   let label = (*case d of*
     TD ⇒ ''*T*'' |
     FD ⇒ ''*F*'' |
     (*IFD v* - -) ⇒ *string-of-nat v*);

```
  return (label, son, son @ ''[label='' @ label @ ''];
'')
}

definition graphifyci1 bdd a ≡ do {
  let ((f,t),y) = a;
  let c = (string-of-nat f @ '' −> '' @ string-of-nat t);
  return (c @ (case y of 0 ⇒ '' [style=dotted]'' | Suc - ⇒ '''') @ '';
'')
}

definition trd = snd ∘ snd
definition fstp = apsnd fst

definition the-thing-By f l = (let
  nub = remdups (map fst l) in
  map (λe. (e, map snd (filter (λg. (f e (fst g))) l))) nub)
definition the-thing = the-thing-By (=)


definition graphifyci :: string ⇒ nat ⇒ bddi ⇒ string Heap where
graphifyci name ep bdd ≡ do {
  s ← serializeci ep bdd;
  let e = map fst s;
  l ← mapM (labelci bdd) (rev (remdups (map fst e @ map snd e)));
  let grp =  (map (λl. foldr (λa t. t @ a @ '';'') (snd l) ''{rank=same;'' @ ''}
'') (the-thing (map fstp l)));
  e ← mapM (graphifyci1 bdd) s;
  let emptyhlp = (case ep of 0 ⇒ ''F;
'' | Suc 0 ⇒ ''T;
'' | - ⇒ '''');
  return (''digraph '' @ name @ '' {
'' @ concat (map trd l) @ concat grp @ concat e @ emptyhlp @ ''}'')
}

end
```

11 Collapsing the levels

theory *Level-Collapse*
imports *Conc-Impl*
begin

The theory up to this point is implemented in a way that separated the different aspects into different levels. This is highly beneficial for us, since it allows us to tackle the difficulties arising in small chunks. However, exporting this to the user would be highly impractical. Thus, this theory collapses all the different levels (i.e. refinement steps) and relates the computations in the heap monad to *boolfunc*.

definition *bddmi-rel cs* ≡ {(*a*,*c*)|*a b c*. (*a*,*b*) ∈ *bf-ifex-rel* ∧ (*c*,*b*) ∈ *Rmi cs*}
definition *bdd-relator* :: (*nat boolfunc* × *nat*) *set* ⇒ *bddi* ⇒ *assn* **where**
bdd-relator p s ≡ ∃$_A$*cs*. *is-bdd-impl cs s* ∗ ↑(*p* ⊆ (*bddmi-rel cs*) ∧ *bdd-sane cs*) ∗ *true*

The *assn* predicate *bdd-relator* is the interface that is exposed to the user. (The contents of the definition are not exposed.)

lemma *bdd-relator-mono*[*intro*!]: *q* ⊆ *p* ⟹ *bdd-relator p s* ⟹$_A$ *bdd-relator q s*
unfolding *bdd-relator-def* **by** *sep-auto*

lemma *bdd-relator-absorb-true*[*simp*]: *bdd-relator p s* ∗ *true* = *bdd-relator p s* **unfolding** *bdd-relator-def* **by** *simp*

thm *bdd-relator-def*[*unfolded bddmi-rel-def*, *simplified*]
lemma *join-hlp1*: *is-bdd-impl a s* ∗ *is-bdd-impl b s* ⟹$_A$ *is-bdd-impl a s* ∗ *is-bdd-impl b s* ∗ ↑(*a* = *b*)
 apply *clarsimp*
 apply(*rule preciseD*[**where** *p*=*s* **and** *R*=*is-bdd-impl* **and** *F*=*is-bdd-impl b s* **and** *F'*=*is-bdd-impl a s*])
  apply(*rule is-bdd-impl-prec*)
 apply(*unfold mod-and-dist*)
 apply(*rule conjI*)
  apply *assumption*
 apply(*simp add*: *star-aci*(*2*))
done

lemma *join-hlp*: *is-bdd-impl a s* ∗ *is-bdd-impl b s* = *is-bdd-impl b s* ∗ *is-bdd-impl a s* ∗ ↑(*a* = *b*)
 apply(*rule ent-iffI*[*rotated*])
  apply(*simp*; *fail*)
 apply(*rule ent-trans*)
  apply(*rule join-hlp1*)
 apply(*simp*; *fail*)
 done

lemma *add-true-asm*:
 assumes <*b* ∗ *true*> *p* <*a*>$_t$
 shows <*b*> *p* <*a*>$_t$
 apply(*rule cons-pre-rule*)
  prefer *2*
  apply(*rule assms*)
 apply(*simp add*: *ent-true-drop*)
 done

lemma *add-anything*:
 assumes <*b*> *p* <*a*>
 shows <*b* ∗ *x*> *p* <λ*r*. *a r* ∗ *x*>$_t$
proof −

note [*sep-heap-rules*] = *assms*
show *?thesis* **by** *sep-auto*
qed

lemma *add-true*:
assumes <*b*> *p* <*a*>$_t$
shows <*b* ∗ *true*> *p* <*a*>$_t$
using *assms add-anything*[**where** *x*=*true*] **by** *force*

definition *node-relator* **where** *node-relator x y* ⟷ *x* ∈ *y*

sep-auto behaves sub-optimal when having (*bf*, *bdd*) ∈ *computed-pointer-relation* as assumption in our cases. Using *node-relator* instead fixes this behavior with a custom solver for *simp*.

lemma *node-relatorI*: *x* ∈ *y* ⟹ *node-relator x y* **unfolding** *node-relator-def* **.**
lemma *node-relatorD*: *node-relator x y* ⟹ *x* ∈ *y* **unfolding** *node-relator-def* **.**

ML‹*fun TRY′ tac = tac ORELSE′ K all-tac*›

```
setup ‹map-theory-simpset (fn ctxt =>
  ctxt addSolver (Simplifier.mk-solver node-relator
    (fn ctxt => fn n =>
      let
        val tac =
          resolve-tac ctxt @{thms node-relatorI} THEN′
          REPEAT-ALL-NEW (resolve-tac ctxt @{thms Set.insertI1 Set.insertI2})
THEN′
          TRY′ (dresolve-tac ctxt @{thms node-relatorD} THEN′ assume-tac ctxt)
      in
        SOLVED′ tac n
      end))
)›
```

This is the general form one wants to work with: if a function on the bdd is called with a set of already existing and valid pointers, the arguments to the function have to be in that set. The result is that one more pointer is the set of existing and valid pointers.

thm *iteci-rule*[*THEN mp*] *mi.ite-impl-R ifex-ite-rel-bf*

lemma *iteci-rule*[*sep-heap-rules*]:
⟦*node-relator* (*ib*, *ic*) *rp*; *node-relator* (*tb*, *tc*) *rp*; *node-relator* (*eb*, *ec*) *rp*⟧ ⟹
<*bdd-relator rp s*>
iteci-lu ic tc ec s
<λ(*r*,*s′*). *bdd-relator* (*insert* (*bf-ite ib tb eb*,*r*) *rp*) *s′*>
apply(*unfold bdd-relator-def node-relator-def*)
apply(*intro norm-pre-ex-rule*)
apply(*clarsimp*)

```
  apply(unfold bddmi-rel-def)
  apply(drule (1) rev-subsetD)+
  apply(clarsimp)
  apply(drule (3) mi.ite-impl-lu-R[where ii=ic and ti=tc and ei=ec, unfolded
in-rel-def])
  apply(drule ospecD2)
  apply(clarsimp simp del: ifex-ite.simps)
  apply(rule cons-post-rule)
   apply(rule cons-pre-rule[rotated])
    apply(rule iteci-lu-rule[THEN mp, THEN add-true])
    apply(assumption)
   apply(sep-auto; fail)
  apply(clarsimp simp del: ifex-ite.simps)
  apply(rule ent-ex-postI)
  apply(subst ent-pure-post-iff)
  apply(rule conjI[rotated])
   apply(sep-auto; fail)
  apply(clarsimp simp del: ifex-ite.simps)
  apply(rule conjI[rotated])
   apply(force simp add: mi.les-def)
  apply(rule exI)
  apply(rule conjI)
   apply(erule (2) ifex-ite-opt-rel-bf[unfolded in-rel-def])
  apply assumption
done

lemma tci-rule[sep-heap-rules]:
<bdd-relator rp s>
  tci s
<λ(r,s'). bdd-relator (insert (bf-True,r) rp) s'>
  apply(unfold bdd-relator-def)
  apply(intro norm-pre-ex-rule)
  apply(clarsimp)
  apply(frule mi.Timpl-rule)
  apply(drule ospecD2)
  apply(clarify)
  apply(sep-auto)
   apply(unfold bddmi-rel-def)
   apply(clarsimp)
  apply(force simp add: mi.les-def)
done

lemma fci-rule[sep-heap-rules]:
<bdd-relator rp s>
  fci s
<λ(r,s'). bdd-relator (insert (bf-False,r) rp) s'>
  apply(unfold bdd-relator-def)
  apply(intro norm-pre-ex-rule)
  apply(clarsimp)
```

apply(*frule mi.Fimpl-rule*)
apply(*drule ospecD2*)
apply(*clarify*)
apply(*sep-auto*)
apply(*unfold bddmi-rel-def*)
apply(*clarsimp*)
apply(*force simp add*: *mi.les-def*)
done

IFC/ifmi/ifci require that the variable order is ensured by the user. Instead of using ifci, a combination of litci and iteci has to be used.

lemma [*sep-heap-rules*]:
⟦(*tb*, *tc*) ∈ *rp*; (*eb*, *ec*) ∈ *rp*⟧ ⟹
<*bdd-relator rp s*>
ifci v tc ec s
<λ(*r*,*s'*). *bdd-relator* (*insert* (*bf-if v tb eb*,*r*) *rp*) *s'*>

This probably doesn't hold.

oops

lemma *notci-rule*[*sep-heap-rules*]:
assumes *node-relator* (*tb*, *tc*) *rp*
shows <*bdd-relator rp s*> *notci tc s* <λ(*r*,*s'*). *bdd-relator* (*insert* (*bf-not tb*,*r*) *rp*) *s'*>
using *assms*
by(*sep-auto simp*: *notci-def*)

lemma *cirules1*[*sep-heap-rules*]:
assumes *node-relator* (*tb*, *tc*) *rp* *node-relator* (*eb*, *ec*) *rp*
shows
<*bdd-relator rp s*> *andci tc ec s* <λ(*r*,*s'*). *bdd-relator* (*insert* (*bf-and tb eb*,*r*) *rp*) *s'*>
<*bdd-relator rp s*> *orci tc ec s* <λ(*r*,*s'*). *bdd-relator* (*insert* (*bf-or tb eb*,*r*) *rp*) *s'*>
<*bdd-relator rp s*> *biimpci tc ec s* <λ(*r*,*s'*). *bdd-relator* (*insert* (*bf-biimp tb eb*,*r*) *rp*) *s'*>
<*bdd-relator rp s*> *xorci tc ec s* <λ(*r*,*s'*). *bdd-relator* (*insert* (*bf-xor tb eb*,*r*) *rp*) *s'*>

using *assms*
by (*sep-auto simp*: *andci-def orci-def biimpci-def xorci-def*)+

lemma *cirules2*[*sep-heap-rules*]:
assumes *node-relator* (*tb*, *tc*) *rp* *node-relator* (*eb*, *ec*) *rp*
shows
<*bdd-relator rp s*> *nandci tc ec s* <λ(*r*,*s'*). *bdd-relator* (*insert* (*bf-nand tb eb*,*r*) *rp*) *s'*>
<*bdd-relator rp s*> *norci tc ec s* <λ(*r*,*s'*). *bdd-relator* (*insert* (*bf-nor tb eb*,*r*) *rp*) *s'*>

```
  using assms
  by(sep-auto simp: nandci-def norci-def)+

lemma litci-rule[sep-heap-rules]:
  <bdd-relator rp s> litci v s <λ(r,s′). bdd-relator (insert (bf-lit v,r) rp) s′>
  apply(unfold litci-def)
  apply(subgoal-tac ‹⋀t ab bb. — introducing some vars ...
        <bdd-relator (insert (bf-False, ab) (insert (bf-True, t) rp)) bb ∗ true>
          ifci v t ab bb
        <λr. case r of (r, x) ⇒ bdd-relator (insert (bf-lit v, r) rp) x>›)
   apply(sep-auto; fail)
  apply(rename-tac tc fc sc)
  apply(unfold bdd-relator-def[abs-def])
  apply(clarsimp)
  apply(intro norm-pre-ex-rule)
  apply(clarsimp)
  apply(unfold bddmi-rel-def)
  apply(clarsimp simp only: bf-ifex-rel-consts-ensured)
  apply(frule mi.IFimpl-rule)
    apply(rename-tac tc fc sc sm a aa b ba fm tm)
    apply(thin-tac (fm, Falseif) ∈ Rmi sm)
    apply(assumption)
   apply(assumption)
  apply(clarsimp)
  apply(drule ospecD2)
  apply(clarify)
  apply(sep-auto)
  apply(force simp add: mi.les-def)
done

lemma tautci-rule[sep-heap-rules]:
 shows node-relator (tb, tc) rp ⟹ <bdd-relator rp s> tautci tc s <λr. bdd-relator
rp s ∗ ↑(r ⟷ tb = bf-True)>
  apply(unfold node-relator-def)
  apply(unfold tautci-def)
  apply(unfold bdd-relator-def)
  apply(intro norm-pre-ex-rule; clarsimp)
  apply(unfold bddmi-rel-def)
  apply(drule (1) rev-subsetD)
  apply(clarsimp)
  apply(rename-tac sm ti)
  apply(frule (1) mi.DESTRimpl-rule; drule ospecD2; clarify)
  apply(sep-auto split: ifex.splits)
done

lemma emptyci-rule[sep-heap-rules]:
  shows <emp> emptyci <λr. bdd-relator {} r>
by(sep-auto simp: bdd-relator-def)
```

lemmas [*simp*] = *bf-ite-def*

Efficient comparison of two nodes.

definition *eqci a b* ≡ *return* (*a* = *b*)

lemma *iteeq-rule*[*sep-heap-rules*]:
⟦*node-relator* (*xb*, *xc*) *rp*; *node-relator* (*yb*, *yc*) *rp*⟧ ⟹
<*bdd-relator rp s*>
  eqci xc yc
<λ*r*. ↑(*r* ⟷ *xb* = *yb*)>$_t$
  apply(*unfold bdd-relator-def node-relator-def eqci-def*)
  apply(*intro norm-pre-ex-rule*)
  apply(*clarsimp*)
  apply(*unfold bddmi-rel-def*)
  apply(*drule* (*1*) *rev-subsetD*)+
  apply(*rule return-cons-rule*)
  apply(*clarsimp*)
  apply(*rule iffI*)
   using *bf-ifex-eq mi.cmp-rule-eq* **apply**(*blast*)
  using *bf-ifex-eq mi.cmp-rule-eq* **apply**(*blast*)
done

end

12 Tests and examples

theory *BDD-Examples*
imports *Level-Collapse*
begin

Just two simple examples:

lemma <*emp*> *do* {
  s ← *emptyci*;
  (*t*,*s*) ← *tci s*;
  tautci t s
} <λ*r*. ↑(*r* = *True*)>$_t$
by *sep-auto*

lemma <*emp*> *do* {
  s ← *emptyci*;
  (*a*,*s*) ← *litci 0 s*;
  (*b*,*s*) ← *litci 1 s*;
  (*c*,*s*) ← *litci 2 s*;
  (*t1i*,*s*) ← *orci a b s*;
  (*t1*,*s*) ← *andci t1i c s*;
  (*t2i1*,*s*) ← *andci a c s*;

(*t2i2*,*s*) ← *andci b c s*;
(*t2*,*s*) ← *orci t2i1 t2i2 s*;
eqci t1 t2
} $<\uparrow>_t$
by *sep-auto*

end

References

[1] K. S. Brace, R. L. Rudell, and R. E. Bryant. Efficient implementation of a BDD package. In *Proceedings of the 27th ACM/IEEE design automation conference*, pages 40–45. ACM, 1991.

[2] M. Giorgino and M. Strecker. Correctness of pointer manipulating algorithms illustrated by a verified BDD construction. In *FM 2012: Formal Methods*, pages 202–216. Springer, 2012.

[3] T. Nipkow. Boolean Expression Checkers. *Archive of Formal Proofs*, June 2014. http://isa-afp.org/entries/Boolean_Expression_Checkers.shtml, Formal proof development.

[4] V. Ortner and N. Schirmer. BDD Normalisation. *Archive of Formal Proofs*, Feb. 2008. http://isa-afp.org/entries/BDD.shtml, Formal proof development.